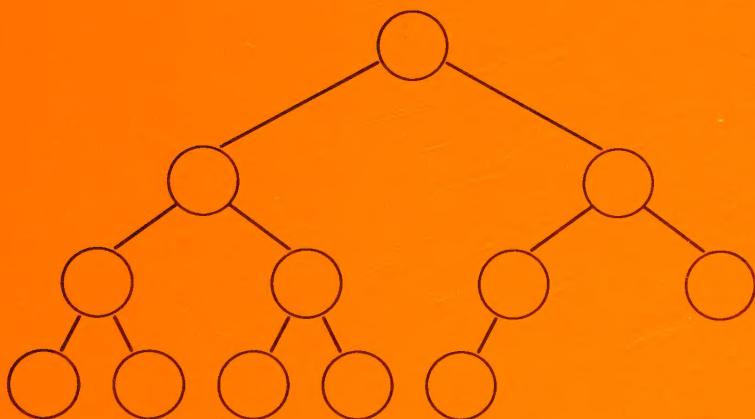


HEAPS



Svante Carlsson

Department of Computer Sciences

Lund University, Lund, Sweden

1986



Digitized by the Internet Archive
in 2026

https://archive.org/details/IA41552418_0008

Acknowledgements

There are many persons who helped me during the development of this thesis. First I would like to thank dr Ferenc Belik for his support and friendship during the days and late nights when we were working on our thesis. I would also like to thank dr Sten Henriksson who first got me interested in data structures and algorithms, and who gave a lot of help and encouragement and worked as a supervisor for me especially in the beginning of the work. In the last year dr Rolf Karlssons help has been very valuable for me. He made me believe in my ideas, and he helped me a lot in the preparation of the thesis.

I would also like to thank the rest of the group for algorithm theory in Lund, and then especially Arne Andersson, Christian Collberg, Christer Mattsson, and Chen Jingsen who all helped me in one way or another. Sonja Hensel saw to that everything around me worked the way it should.

I would also like to thank my parents who encouraged me to study, and gave me all possible help to do so. Eva Carlsson, my children Hanna and Viktor who had to put up with a father often working late at night, and my comfort when times were hard Lotta Nord are also worth mentioning.

Table of Contents

1. Introduction	
1.1 History and Motivation	1
1.2 Major Contributions	2
1.3 Notations and Basic Definitions	2
1.4 Standard Algorithms	4
2. Modification of the Heap Structure	
2.1 Improvement by Scattering the leaves	6
2.2 Improvement by Using a Ternary Heap	7
2.3 A Comment on Optimality	9
2.4 Some Comments on the Application of Heaps	10
3. Improved Rearrangement Procedure in the Worst Case	
3.1 The Binary Search HEAPSORT Algorithm	11
3.2 Analysis of the Algorithm	12
3.3 Further Improvements	15
3.4 An Example of the Further Improved Algorithm	16
3.5 Improvements on HEAPSORT	17
4. Average-Case Results on Heaps	
4.1 An Improved Rearrangement Procedure for the Average Case	18
4.2 Average-Case Analysis of the New Rearrangement Algorithm	20
4.3 An Average-Case Analysis of the New HEAPSORT	31
4.4 Analysis of the Floyd-Williams HEAPSORT Algorithm	34
4.5 Average Number of Interchanges	35
4.6 Simulation Results	36
4.7 Conclusions	36
5. Amortized Cost of a Priority Queue Structure	
5.1 A New Data Structure with Good Amortized Behavior	38
5.2 Analysis of the Amortized Cost	39
5.3 Comments	40
6. The Deap — a Double-Ended Heap	
6.1 Introduction to Priority Dequeues	44
6.2 The Data Structure	45
6.3 Algorithms for the Operations	46
6.4 Complexity Analysis	47
6.5 Concluding Remarks	48
7. Conclusions and Further Research	49
References	51

Appendices

Appendix A

A.1	The Algorithm for HEAPSORT Using a Complete Binary Heap	53
A.2	The Algorithm for HEAPSORT Using a Complete Ternary Heap	54
A.3	The Algorithm for HEAPSORT Using a Binary Heap with Scattered Leaves	56
A.4	The Algorithm for HEAPSORT Using a Ternary Heap with Scattered Leaves	59

Appendix B

B.1	The Algorithm for Inserting an Element in a Complete Binary Heap	62
B.2	The Algorithm for Inserting an Element in a Ternary Heap with Scattered Leaves	63

Appendix C

C.1	AHEAPSORT with Good Worst-Case Behavior	64
-----	---	----

Appendix D

D.1	A Rearrangement Procedure for Heaps with a Good Average-Case Behavior	66
-----	---	----

Appendix E

E.1	Insertion and Delete-Min Algorithms for a Heap with a Good Amortized Behavior	67
-----	---	----

Appendix F

F.1	Implementations of the Operations on a Deap	69
-----	---	----

1. Introduction

1.1 History and Motivation

This thesis deals with the well-known data structure called heap. This structure was first introduced 1964 by Williams [15] in a sorting algorithm called HEAPSORT, which sorts in $O(n \log n)$ time without using extra space. Later that year Floyd improved the algorithm by constructing a heap in linear time [6]. It is not difficult to analyse the worst-case behavior of the algorithm, but the average number of operations has not yet been analysed. Doberkat [2], however, analysed the number of comparisons and the number of data movements of Floyd's heap construction algorithm.

The heap can also be used to implement priority queues, where it is possible to insert an element, and to find and delete the element with the highest priority in $O(\log n)$ operations. Sometimes it is also interesting to create a heap of a number of elements given at the same time, which can be done in linear time. The most natural insertion algorithm works in $O(\log n)$ time in the worst case, and in constant time on the average, as shown by Porter and Simon [13]. Gonnet and Munro [10] gave an algorithm for insertion which uses only $O(\log \log n)$ comparisons in the worst case, while the number of data movements still are $O(\log n)$. Deletion of the element with highest priority is normally done with at most $2 \log n$ comparisons and $\log n$ data movements in both worst and average case. Gonnet and Munro, however, have an algorithm which only uses $\log n + g(n) - \epsilon(n)$ comparisons, where $g(n)$ is defined as 0 if $n \leq 1$ and $g(\log n) + 1$ otherwise, and $\epsilon(n)$ is between 0 and 1. When a heap is created from a number of element Floyd's heap creation algorithm can preferably be used, even if it takes $2n$ comparisons and n data movements in the worst case and $1.88n$ comparisons and $0.744n$ data movements on the average, while Gonnet and Munro presented an algorithm using at most $1.625n$ comparisons and $1.54 n$ on the average, but also extra space or many data movements, and another algorithm using at most $1.77n$ comparisons and no extra space.

The thesis deals only with implicit heaps related to the heap proposed by Williams, and not with any other form of priority queues which are sometimes also called heaps. However, in Chapter 5 a priority queue, which is based on this structure but uses extra space for information about the structure to get a good amortized cost, is introduced. Various modifications of the data structure are presented in Chapter 2, and their effect on the number of comparisons in HEAPSORT is analysed.

In the following two chapters improvements of the algorithm for deleting the element with highest priority are introduced and analysed. This improves the behavior of both the heap construction and HEAPSORT algorithm. In Chapter 3 the worst case behavior is in focus and in Chapter 4 the average case behavior. The average case analysis of the improvement in Chapter 4 gives a method to analyse the average case of old heap algorithms of Williams, including HEAPSORT.

As mentioned, Chapter 5 introduces a data structure for a priority queue with good amortized behavior which also can be used to get a good sorting algorithm, while Chapter 6 contains a new way of implementing a double-ended priority queue, a so called priority deque (where it is possible to delete both the element with highest priority as well as the one with the lowest, in $O(\log n)$ time). This structure is an improvement of the MinMaxHeap presented by Atkinson et al. [1], using the ideas from the new heap algorithms together with some observations about the structure.

Finally the algorithms of the thesis are collected in the Appendices. Some of the algorithms are written in Pascal while others are just given in pseudo code. The most important algorithms are also included in the text.

1.2 Major contributions

The major contributions of this thesis concerning the improvements on HEAPSORT and the operations on the implicit priority queue are summarized in Table 1.1. The average case analysis in Chapter 4 is tighter than earlier ones, and the average cost of HEAPSORT is now almost analysed. The worst number of comparison to delete the element with highest or lowest priority from the same structure is reduced from $1.5 \log n$ to just $\log n$, while the structure is created in $2.07 n$ comparisons instead of, as earlier $2.15 n$. A new data structure with a good amortized cost is also presented.

Operation on the heap	Previous best result	New result	Comments
Insertion			
worst case	$\log \log n$ [10]		No improvements made
average case	2.61 [13]		No improvements made
Delete after priority			
worst case	$\log n + g(n) - \epsilon(n)$ [10]	$\lfloor \log n - 1 \rfloor + \gamma(\log \frac{n-1}{2}) - 1$	Chapter 3
average case	$\log n + g(n) - \epsilon(n)$ [10]	$\lfloor \log n \rfloor + 0.329 + M(n)$	Chapter 4
Create			
worst case	1.77 n [10]	1.77 n	Chapter 3
average case	1.77 n [10]	1.65 n	Chapter 4
HEAPSORT			
worst case	$2 n \log n + O(n)$ [6]	$n \log n + n \gamma(n) + O(n)$	Chapter 3
average case	$2 n \log n - 3.025 n$ [9]	$n \log n + 0.700 n$	Old result by simulation Chapter 4

$g(n)$ is defined as 0 if $n \leq 1$ and $1 + g(\log n)$ otherwise.

$\gamma(n)$ is defined as the inverse of $\Gamma(n)$, where $\Gamma(n) = 1$ if $n = 1$ and $\Gamma(n-1) + 2^{\Gamma(n-1)-1+n-2}$ otherwise.

$\epsilon(n)$ and $M(n)$ are functions in the range from 0 to 1.

Table 1.1: Table of the number of comparisons for various heap operations, where New result means result presented in this thesis.

1.3 Notations and Basic Definitions

Since the whole thesis is about heaps, a definition of heaps and the concepts is in its place.

Heap shape: A binary tree where all leaves are at at most two levels and are stored as far to the left as possible.

Heap order: A binary tree where an element has higher or equal priority to its sons and lower or equal priority to its father.

Heap: A heap is an implicit data structure where all elements are stored in an array, and no extra space is used. The elements form a binary tree with heap shape and heap order. An element at position k has its sons, if any, at positions $2k$ and $2k+1$ and its father at position $k \div 2$. In Figure 2.1 an example of a heap is shown.

Min-heap: A heap where small key value corresponds to high priority.

Max-heap: A heap where large key value corresponds to high priority.

Level in a heap: All element at the same distance from the root forms one level in the heap.

Height of a heap: The number of levels in a heap.

Rearrangement of a heap: When the root of the heap is the only element that violates the heap order, it has to be rearranged so that heap order is reestablished.

Last leaf: The rightmost leaf at the lowest level of the heap is called the last leaf. It is also the element at the last occupied position of the array storing the heap.

Path of maximum sons: The path of maximum sons is recursively defined as the son of the root with the largest key value in a max-heap concatenated with the path of maximum sons for that son.

Path of minimum sons: The path of minimum sons is defined for a min-heap in a similar way as the path of maximum sons for a max-heap, with the obvious changes.

It is also worth mentioning that all logarithms are to base two if not stated otherwise.

1.4 Standard Algorithms

The standard algorithms on heaps are as follows:

Insertion: Insert the new element at the first free position of the array. Let it repeatedly swap place with its father until it has lower priority than its father.

Rearrangement: Since heap order is violated, the element stored at the root is compared with its sons and swapped with the one with highest priority. The element is repeatedly compared with its new sons until it has found its place and the heap order is not violated.

Deletion of the element with highest priority: Remove the element at the top of the heap, and replace it with the last leaf. Now it only remains to rearrange the heap as described above.

Creation: Place all the elements in an array and treat it as a binary tree with heap shape. Start with the last element that has a son. We can regard this as a heap where the sons of the root are roots of smaller subheaps but where the root itself might violate the heap order, so we can perform a rearrangement on this heap. This is done for all elements up to the root, that is all element stored before that element in the array, and then we have created a heap of all the elements.

Sorting: Create a heap. Then delete the element with highest priority repeatedly until all elements are removed in priority order. The removed element is stored in the position freed when the last leaf is moved up. The elements will thus be sorted in the array.

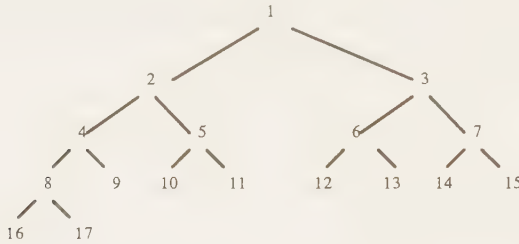
2. Modifications of the Heap Structure

Contents

It is possible to improve the worst-case behavior of operations on heaps, when counting number of comparisons between elements in the heap, by modifying the traditional binary heap. The improvement suggested is the use of a heap with "scattered" leaves. This method can be combined with the use of a ternary heap. Some other improvements of worst-case behavior compared to standard algorithms are discussed at the end of the chapter.

2.1 Improvement by Scattering the Leaves

A complete binary heap stores its elements in the following order:

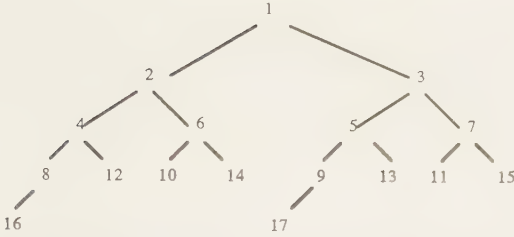


where vertex m has the sons $2m$ and $2m + 1$. The height of a heap with n vertices is $\lfloor \log_2 n \rfloor$.

Let h be equal to the height of the heap, then the number of comparisons necessary in the worst case (C_n) is $2h$, except when there is only a single leaf at the lowest level. Then $C_n = 2h - 1$. Hence,

$$C_n = \begin{cases} 2 \lfloor \log_2 n \rfloor - 1 & \text{if } n \text{ is a power of 2} \\ 2 \lfloor \log_2 n \rfloor & \text{otherwise} \end{cases}$$

Another possible way to implement a heap is to use a binary heap with scattered leaves, which stores its elements in the following order:



where vertex m has the sons $m + a$ and $m + 2a$, where $a = 2^{\lfloor \log_2 m \rfloor}$. The height of a heap with n elements is $\lfloor \log_2 n \rfloor$.

Theorem 2.1: The number of comparisons to rearrange a binary heap with scattered leaves is sometimes less in the worst-case than for a complete binary heap, but never worse.

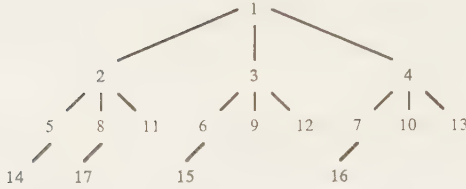
Proof: Let C_n and h have the same meaning as before, and let C_n' be the number of comparisons necessary in the worst case for a binary heap with scattered leaves. Then $C_n' = 2h$ if the bottom row is more than "half-full", else $C_n' = 2h - 1$.

$$C_n' = \begin{cases} 2 \lfloor \log_2 n \rfloor - 1 & \text{if } n < 1.5 \cdot 2^{\lfloor \log_2 n \rfloor} \\ 2 \lfloor \log_2 n \rfloor & \text{if } n \geq 1.5 \cdot 2^{\lfloor \log_2 n \rfloor} \end{cases}$$

Since $n < 1.5 \cdot 2^{\lfloor \log_2 n \rfloor}$ if n is a power of 2, C_n is always greater than or equal to C_n' .

2.2. Improvement by Using a Ternary Heap

It is a known fact that a ternary heap has a better asymptotic worst-case behavior than binary heap (cf. [11], Exs. 5.2.3-18, 5.2.3-28), and it is possible to construct a ternary heap with scattered leaves which stores its elements in the following order:



where vertex m has the sons $m + a$, $m + 2a$, and $m + 3a$ where $a = 3^{\lfloor \log_3 2m \rfloor}$. The height of a heap with n elements is $\lfloor \log_3 2n \rfloor$.

Theorem 2.2: A ternary heap with scattered leaves is rearranged with asymptotically fewer comparisons in the worst-case than a binary heap with scattered leaves.

Proof: Let C_n , C_n' and h have the same meaning as before, and let C_n'' be the number of comparisons necessary in the worst case for a ternary heap with scattered leaves. Then $C_n'' = 3h - 2$ if the bottom row is less than one-third filled, $C_n'' = 3h - 1$ if the bottom row is filled to between one and two thirds, $C_n'' = 3h$ otherwise.

$$C_n'' = \begin{cases} 3 \lfloor \log_3 2n \rfloor - 2 & \text{if } n < (5/6) 3^{\lfloor \log_3 2n \rfloor} \\ 3 \lfloor \log_3 2n \rfloor - 1 & \text{if } (5/6) 3^{\lfloor \log_3 2n \rfloor} < n < (7/6) 3^{\lfloor \log_3 2n \rfloor} \\ 3 \lfloor \log_3 2n \rfloor & \text{if } n > (7/6) 3^{\lfloor \log_3 2n \rfloor} \end{cases}$$

Since C_n' for a binary heap with scattered leaves is never less than $2 \lfloor \log_2 n \rfloor - 1$, and C_n'' for a ternary heap with scattered leaves never exceeds $3 \lfloor \log_3 2n \rfloor$ it is only left to show that

$$3 \lfloor \log_3 2n \rfloor < 2 \lfloor \log_2 n \rfloor - 1, \text{ for all } n > X.$$

which is obviously true for some X , since $3 \log_3 n < 2 \log_2 n$.

An investigation shows that a ternary heap with scattered leaves is rearranged with fewer comparisons in the worst-case than a binary heap with scattered leaves for all values greater than or equal to 98304, and is never worse for values greater than 383.

It is also possible to see that if one has to rearrange a heap for all values less than or equal to n — as in HEAPSORT — it is favorable to use a ternary heap with scattered leaves for all $n > 259$, since the number of comparisons gained when using a binary heap with scattered leaves (one for the values 5, 11, 14, 15, 23, 41-47, 95, 122-127 - totally 19) is less than the number of comparisons gained when using a ternary heap with scattered leaves (one for values 64-67, 192-202, 256-260 — totally 20). See also Table 2.1.

Number of elements	Binary case better	Ternary case better	Equal
1 - 100	13.0%	4.0%	83.0%
101 - 1000	2.5%	21.8%	75.7%
1001 - 10000	0.0%	56.0%	44.0%
10001 - 100000	0.0%	73.1%	26.9%
100001 -	0.0%	100.0%	0.0%

Table 2.1: Comparison between the number of comparisons needed in the worst case for binary and ternary heaps with scattered leaves.

2.3. A Comment on Optimality

It is easy to show that a k -ary heap requires at least $k \lfloor \log_k (k-1)n \rfloor - (k-1)$ comparisons to rearrange the heap in the worst case. To find the integer that minimizes this function is equal to finding the integer that minimizes $k/\ln k$, which is 3. This shows us that the ternary heap has the asymptotically best worst-case behavior.

This result agrees with what was shown by Göbel and Hoede [8], when they let the cost of a node to be equal to the number of sons of a node.

2.4. Some Comments on the Applications of Heaps

It has been shown earlier that a ternary heap has a better asymptotic worst-case behavior than a binary heap. The asymptotic improvement on the number of comparisons is approximately 5%, and the improvement on the number of operations is approximately 14% (cf. [11], Exs. 5.2.3-18, 5.2.3-28). Both the binary and the ternary heap with scattered leaves can be used in the two major applications of heaps, namely in priority queues and in HEAPSORT (see Appendix A and B). It can also be combined with other suggested improvements of operations on heaps, for instance the use of a number of separate heaps instead of one (cf. [5]).

It is possible to save some comparisons, when rearranging a heap after deleting the top element, as done in HEAPSORT and in priority queues, by keeping track of the path of ancestors of an element, since at least one unnecessary comparison between an element and its ancestor is done after every deletion.

3. Improved Rearrangement Procedure in the Worst Case

Contents

Two algorithms which asymptotically halves the number of comparisons made by the common HEAPSORT, are presented and analysed in the worst case. One algorithm is of practical interest, where the number of comparisons is shown to be $(n+1)(\log(n+1) + \log\log(n+1) + 1.82) + O(\log n)$ in worst case to sort n elements, without using any extra space, and one of more theoretical interest with a worst case of $n \log n + n \gamma(n) + O(n)$ comparisons, where $\gamma(n)$ is defined as the invers of $\Gamma(n)$ which is defined as 1 if $n=1$ and $\Gamma(n-1) + 2^{\Gamma(n-1)-1+n-2}$ otherwise. Both methods improve the sorting by improving the rearrangement and thus also the deletion of the element with the highest priority.

3.1. The Binary Search HEAPSORT Algorithm

The critical part of the HEAPSORT algorithm is the rearrangement procedure, where a leaf and the root in the heap change places. The former leaf is swapped with the largest of its new sons, and so on, until it is larger than both sons or it is a leaf. To do this, two comparisons are made at each level of the heap below the root, and the number of levels is at most $\lfloor \log k \rfloor$ if k is the number of elements left to be sorted.

In this improved version, no comparisons are made between a node and its children. Instead, a path of maximum sons is found, by letting the larger of the sons of the root be in that path, and connecting it with the path of maximum sons for that node. A binary insertion in such a path can be performed, since any path from the root to a leaf in a heap is an ordered list. The length of that path is at most $\lfloor \log k \rfloor$.

The only part of the HEAPSORT algorithm that has to be changed is the rearrangement procedure. First, a path of maximum sons is found. Choose the element to be inserted as the root of the heap to be rearranged during the creation of the heap, and the last leaf during the sorting. Then, a binary search for the place of the element which has to be inserted in the path, is performed. This can easily be done after observing that any node j has the ancestor $j \text{ div } 2^k$ where k is the length of the path between the node and its ancestor, ($j \text{ div } 2^k$ can be computed by shifting j k steps to the right). Finally all elements that are larger than the last leaf are moved one step up, and the element is

and not on the value of the element, which is to be inserted. In the following Figure the procedure HEAPSORT is given. The auxiliary procedure REARRANGE is at first used for the creation of the heap and then for the necessary rearrangement after removing the current maximum element from the root of the heap.

3.2. Analysis of the Algorithm

Theorem 3.1: The number of comparisons to sort n element with the binary search HEAPSORT algorithm is never more than $(n+1) (\log(n+1) + \log\log(n+1) + 1.82)$.

Proof: The maximum number of comparisons during one call of the REARRANGE procedure is $h + \lfloor \log h \rfloor + 1$, where h is the height of the heap to be rearranged. If the number of elements to be sorted is $2^k - 1$, the number of comparisons between elements can be shown to be:

During the creation of the heap $(n+1)/4$ heaps of height 1, $(n+1)/8$ of height 2, $(n+1)/16$ of height 3, and so on, are rearranged. This gives a total number of:

$$\sum_{k=1}^{\log(n+1)-1} (k + \lfloor \log k \rfloor + 1) \frac{n+1}{2^{k+1}} = (n+1) \left(\sum_{k=1}^{\log(n+1)-1} \frac{k}{2^{k+1}} + \sum_{k=1}^{\log(n+1)-1} \frac{1 + \lfloor \log k \rfloor}{2^{k+1}} \right) \leq$$

$$\leq (n+1 - O(\log n)) + 0.8164 \dots (n+1) \leq 1.82 n - O(\log n)$$

During the sorting $(n+1)/2$ heaps of height $\log(n+1) - 1$, $(n+1)/4$ of height $\log(n+1) - 2$, and so on, are rearranged to give a total of:

$$\sum_{k=1}^{\log(n+1)-1} (\log(n+1) - k + \lfloor \log (\log(n+1) - k) \rfloor + 1) \frac{n+1}{2^k} \leq$$

$$\leq (n+1) \log(n+1) \sum_{k=1}^{\log(n+1)-1} \frac{1}{2^k} - (n+1) \sum_{k=1}^{\log(n+1)-1} \frac{k}{2^k} + (n+1) \log \log(n+1) + (n+1) \sum_{k=1}^{\log(n+1)-1} \frac{1}{2^k} \leq$$

$$\leq (n+1)(\log(n+1) + \log \log(n+1))$$

which will prove the theorem.

```
procedure REARRANGE(var A : vector; X : elementtype; I,N : integer);

var J, K, NrOfLevels,
    Bot, Top, Mid : integer;

begin
  J:= 2*I;

{ *** LINEAR SEARCH *** }
  while J < N do
    if A[ J ] < A[ J+1 ] then
      J:= 2*(J+1)
    else
      J:= 2*J;

  if J > N then
    J:= J div 2;           { J is now a leaf }

{ *** BINARY SEARCH *** }

  NrOfLevels:= log( J div I );
  { Number of levels between I and J }
  TOP:= NrOfLevels;
  BOT:= 0;

  while Top > Bot do
  begin
    Mid:= (Top+Bot) div 2;
    if X ≤ A[ J div 2**Mid ] then
      Top:= Mid
    else
      Bot:= Mid+1
  end;

{ *** INSERTION *** }

  for K:= NrOfLevels - 1 downto Top do
    A[ J div 2**(K+1) ]:= A[ J div 2**K ];

  A[ J div 2**Top ]:= X;

end;
```



```

procedure HEAPSORT(var A : vector; N : integer);
{ *** This procedure is similar to the usual HEAPSORT *** }

var I      : integer;
    Temp   : elementtype;

begin

{ *** HEAP CREATION *** }

    for I:= N div 2 downto 1 do
        REARRANGE(A,A(I),I,N);

{ *** SORTING *** }

    for I:= N-1 downto 1 do
        begin
            Temp:= A[ I+1 ]; A[ I+1 ]:= A[ 1 ];
            REARRANGE(A,Temp,1,I)
        end;

end;

```

Figure 3.1: The HEAPSORT algorithm implemented in PASCAL, where 2^{**k} means 2^k , and $\log(n)$ is a function that returns $\lfloor \log_2 n \rfloor$.

Compared with the average result for QUICKSORT, $> 1.38n(\log n - 2) + \log n$ [9], HEAPSORT now uses fewer comparisons in the worst case than QUICKSORT does in the average case. The result for HEAPSORT is almost as good as that for MERGESORT, which uses an optimal number of comparisons, $n(\log n) - n$. But MERGESORT requires $O(n)$ extra storage, which HEAPSORT does not.

The number of comparisons can be reduced further by using a binary heap with scattered leaves (Chapter 2). The creation of a heap will then take $1.75n - O(\log n)$ comparisons, and the sorting part $n/2$ less than when using a normal heap, if $(n+1)/3$ is a power of 2.

3.3. Further Improvements

It is possible to improve the rearrangement procedure further, also yielding faster sorting algorithms. Gonnet and Munro [10] also use the idea of maximum sons, together with a binary insertion technique, but they improve this by not searching the entire path at once. Instead they find the path of maximum sons down to level $\lfloor \log n \rfloor - \log \log n$, and compare the last leaf with the last element in the path found. If the last leaf is larger then insert it with binary search technique in the path, otherwise repeat the same thing on the rest of the path, i.e. continue to find the rest of the path of maximum sons but stop at the logarithm of the length from the leaf to ensure that you can do the binary search at the same cost as you gain by not finding the whole path. They showed that this algorithm takes $\lfloor \log n \rfloor + g(n)$ comparisons, where $g(n)$ is defined as 0 if $n \leq 1$ and $1 + g(\log n)$ otherwise. They also showed that the technique of comparing the last leaf with an element in the path of maximum sons is optimal in the worst case, and that this result is a lower bound for the delete-max operation.

A new algorithm with an even better worst case behavior uses the same technique as Gonnet and Munro claimed to be optimal, but the selection of places to stop and compare the last leaf with can be further refined.

We know that the last leaf can be inserted at the end of the path of maximum sons. Therefore, it might be compared with the leaf element in the path. If the last leaf is the smallest, the total cost will be one more than the cost to find this path, including earlier stops. If it is not the smallest but we want to have the same cost in this case also, we have to know that the last leaf is smaller than the father of the leaf element in the path, by an earlier comparison. If the last leaf was larger than that element we had saved two comparisons for a binary search in the path above, one to determine which son should be in the path of maximum sons, and one to determine if the last leaf is smaller or larger than the last element in the path. With two comparisons we can insert an element in a list of length three, so we could have inserted the last leaf in the list of elements tree, four, and five from the end of the path of maximum sons. This we could have done at the same cost as for inserting it at the end of the path.

By the same arguments as before, we had to compare the last leaf with the sixth element from the end in the path of maximum sons, to be able to use the algorithm described above. If the last leaf was larger than the sixth element from the end we saved seven comparisons, five for determining the path of maximum sons and two for the stops at element one and two from the end. These comparisons we could have used to insert the last leaf in the list of elements $6+1$ to $6+2^7-1$.

Let us define the function $\Gamma(n)$ to be the places we have to stop on the path of maximum sons and compare with the last leaf, where $\Gamma(1)$ is the last place we have to stop on and so on, then

$$\Gamma(n) = \begin{array}{ll} 1 & \text{if } n = 1 \\ \Gamma(n-1) + 2^{\Gamma(n-1)-1+n-2} & \text{otherwise} \end{array}$$

This function grows faster than $G(n)$ which is the inverse of $g(n)$ which Gonnet and Munro uses, as can be seen in Table 3.1. Let γ be the invers function of Γ , then we can give the cost for deleting the largest element in a heap. Since we know that the last leaf is smaller than the largest son of the root we don't have to compare those two elements, and the total number of comparisons is $\lfloor \log n-1 \rfloor + \gamma(\log \frac{n-1}{2}) - 1$.

Theorem 3.2: A delete-min operation of a heap can be performed in $\lfloor \log n-1 \rfloor + \gamma(\log \frac{n-1}{2}) - 1$ comparisons in the worst case.

Proof: The theorem follows from the reasoning above.

<u>n</u>	<u>$\Gamma(n)$</u>	<u>$G(n)$</u>
1	1	1
2	2	2
3	6	4
4	134	16
5	$134+2^{136}$	65536

Table 3.1: Table over the growth rate of functions $\Gamma(n)$ and $G(n)$.

3.4. An Example of the Further Improved Algorithm

To get a better idea of how the algorithm works we will perform a delete-max operation of a heap with 127 elements using only eight comparisons, (Figure 3.2).

The sons of the root are compared. We know that the last leaf is smaller than the largest of the sons of the root. Find the path of maximum sons down to the level above the last. This will take five comparisons. Compare the last leaf with the lowest element identified in the path of maximum sons, (marked m2 in Figure 3.2). If the last leaf is the largest then insert it among elements m3, m4, and m5 using only two comparisons, which gives a total of eight comparisons. If it is smaller, compare the sons of m2 and compare the largest of them with the last leaf, which also adds up to eight comparisons.

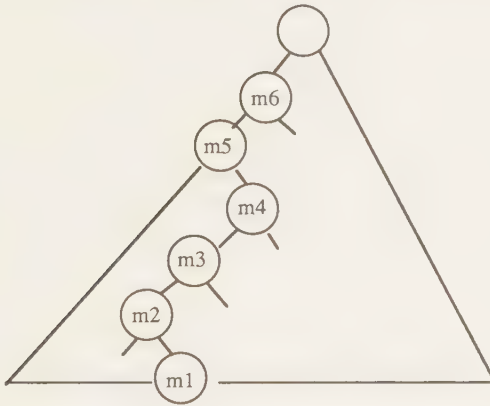


Figure 3.2: A heap with 127 elements with path of maximum sons shown.

3.5. Improvements on HEAPSORT

The above algorithm can also be used for sorting n elements in an array without using any extra space. A heap can be constructed using the technique described by Floyd [6], but with this new algorithm instead of the sift-up algorithm. After that, repeated deletions from the heap are performed.

The heap can be built in linear time, ($1.77 n$ comparisons), and the deletions will take less than $n(\lfloor \log n - 1 \rfloor + \gamma(\log \frac{n-1}{2}) - 1)$ comparisons, giving a total of $n \log n + n \gamma(n) + O(n)$ comparisons.

4. Average-Case Results on Heaps

Contents

A new algorithm for rearranging a heap is presented, and analysed in the average case. The average case upper bound for deleting the maximum element of a random heap is improved, and is shown to be less than $\lfloor \log n \rfloor + 0.329 + M(n)$ comparisons, where $M(n)$ is between 0 and 1. It is also shown that a heap can be constructed using $1.650n + O(\log n)$ comparisons with this algorithm, which is the best result for any algorithm which does not use any extra space. The expected time to sort n elements is argued to be less than $n \log n + 0.700n + O(\log n)$, while simulation result points to an average case of $n \log n + 0.4n$, which will make it the fastest in-place sorting algorithm. The same technique is used to show that the average number of comparisons when deleting the maximum element of a heap using Williams' algorithm for rearrangement is $2(\lfloor \log n \rfloor - 1.329 + L(n))$ where $L(n)$ also is between 0 and 1, and the average cost for Floyd-Williams' HEAPSORT is at least $2n \log n - 3.78n$, counting only comparisons. An analysis of the number of interchanges when deleting the maximum element of a random heap, which is the same for both algorithms, is also presented.

4.1. An Improved Rearrangement Procedure for the Average Case

In Chapter 3 a technique for the rearrangement is used, where a path of maximum sons is found, and the element is inserted in that path. A path of maximum sons of a heap is the largest son of the root concatenated with the path of maximum sons of the subheap with that element as the root. This path is found in $\lfloor \log n \rfloor$ comparisons. To get a good worst-case performance the element stored last in the array can be inserted in this path by binary insertion using only $O(\log \log n)$ comparisons. On the other hand, if the element that has to be inserted in the path of maximum sons is inserted by using linear search from the leaf-end of the path the average cost is bounded by a constant, but the worst-case cost is now $O(\log n)$. The linear search in the path is equivalent to the algorithm for inserting an element in a heap. This yields the rearrangement algorithm in Figure 4.1.

```

procedure rearrange( var A : vector; I,N : integer; X : elementtype),
var J   :   integer;

begin
    { find path of maximum sons }

    J:= 2*I;

    while J < N do
    begin
        if A[ J+1 ] >= A[ J ] then
            J:= J+1;

            A[ J div 2 ]:= A[ J ];
            J:= 2*J
        end;

        if J > N then
            J:= J div 2
        else
            A[ J div 2 ]:= A[ J ];

            { insert in path of maximum sons }

            while J > I do
                if X > A[ J div 2 ] then
                    begin
                        A[ J ]:= A[ J div 2 ];
                        J:= J div 2;
                    end
                else
                    I:= J;
                end

            A[ J ]:= X
        end;
    end;

```

A is the array storing the heap, I is the first and N the last element involved in the rearrangement, and X is the element to be inserted in the path of maximum sons.

Figure 4.1: A rearrangement algorithm for heaps with a good average case performance.

4.2. Average-Case Analysis of the New Rearrangement Algorithm

Porter and Simon showed in [13] that a random element can be inserted in a random heap using 2.61 comparisons on the average. Together with the result of Doberkat [2] which shows that a heap with n elements can be built in $1.88 n$ comparisons on the average we can easily see that we can sort the elements in $n \log n + O(n)$ comparisons. This is done by first building a heap, and then finding a path of maximum sons and inserting the last leaf in that path. Since the last leaf is smaller than an average random element the result follows. More formally we get:

Theorem 4.1: A heap is built with the algorithm in Figure 4.1 using $1.650 n + O(\log n)$ comparisons on the average.

Proof: This theorem is proved by analysing a slightly different algorithm which has the same number of comparisons as the algorithm in Figure 4.1. It also has the same number of interchanges as Floyd's algorithm. This is achieved by not moving any element until its final place after rearrangement is found. This algorithm uses more overhead than the one in Figure 4.1 and will take longer time when executed on a computer.

If we do this we have to first find the path of maximum sons using one comparison for each element in the path (first term), and then, beginning with the leaf, for each element in that path either compare it with the element to be inserted, or move it one step towards the root in the heap. The first element larger than the inserted element, however, is both compared and moved (second term). If the inserted element is the largest then it is already stored at the root, and we save one comparison (third term).

Since we find $(n+1)/4$ paths of length 1, $(n+1)/8$ paths of length 2 and so on, when building a heap, and the probability that the inserted element is the largest is $1/(2^{k+1} - 1)$ if the path length is k , the total number of comparisons and interchanges together, in a heap with $n = 2^h - 1$ elements is:

$$\sum_{k=1}^{h-1} \frac{k(n+1)}{2^{k+1}} + \sum_{k=1}^{h-1} \frac{(k+1)(n+1)}{2^{k+1}} - \sum_{k=1}^{h-1} \frac{1}{2^{k+1} - 1} \frac{n+1}{2^{k+1}} =$$

$$2(n+1) \sum_{k=1}^{h-1} \frac{k}{2^{k+1}} + (n+1) \sum_{k=1}^{h-1} \frac{1}{2^{k+1}} - (n+1) \sum_{k=1}^{h-1} \frac{1}{2^{k+1} (2^{k+1} - 1)} <$$

$$2(n+1) - O(\log n) + \frac{n+1}{2} - O(\log n) - 0.106(n+1) =$$

$$2.394 (n+1) - O(\log n)$$

In [2] it is shown that the number of interchanges when using Floyd's algorithm is $0.744 n + O(\log n)$ on the average, so the number of comparisons is:

$$2.394 (n+1) - O(\log n) - 0.744 n + O(\log n) = 1.650 n + O(\log n).$$

Lemma 4.1: If the path of maximum sons is given, the largest element can be deleted from a random heap of $n=2^k - 1$ elements, in 1.329 comparisons on the average.

Proof: Label the nodes in the path of maximum sons $m_1, m_2, \dots$, and the nodes in the path from the last leaf to the root $l_1, l_2, \dots$, starting at the leaf, and let h be the length of the part of the paths that differs. Call the penultimate leaf Y . Let also P_{ijh} denote the probability that $l_1 < m_1$ for a fixed h . This can also be expressed as the probability that it is sufficient with i comparisons to insert the last leaf in the path of maximum sons for a certain value of h . (See Figure 4.2)

To find P_{ijh} for any i , we look at a fixed h , and then start by finding P_{1jh} :

$$P_{1jh} = P(l_1 < m_1) =$$

$$P(l_1 < m_1 \mid l_2 < m_2) P(l_2 < m_2) + P(l_1 < m_1 \mid l_2 > m_2) P(l_2 > m_2) =$$

$$(P(l_1 < m_1 \mid l_2 < m_2 \wedge l_1 < Y) P(l_1 < Y) +$$

$$P(l_1 < m_1 \mid l_2 < m_2 \wedge l_1 > Y) P(l_1 > Y)) P(l_2 < m_2) +$$

$$(P(l_1 < m_1 \mid l_2 > m_2 \wedge l_1 < Y) P(l_1 < Y) +$$

$$P(l_1 < m_1 \mid l_2 > m_2 \wedge l_1 > Y) P(l_1 > Y)) P(l_2 > m_2)$$

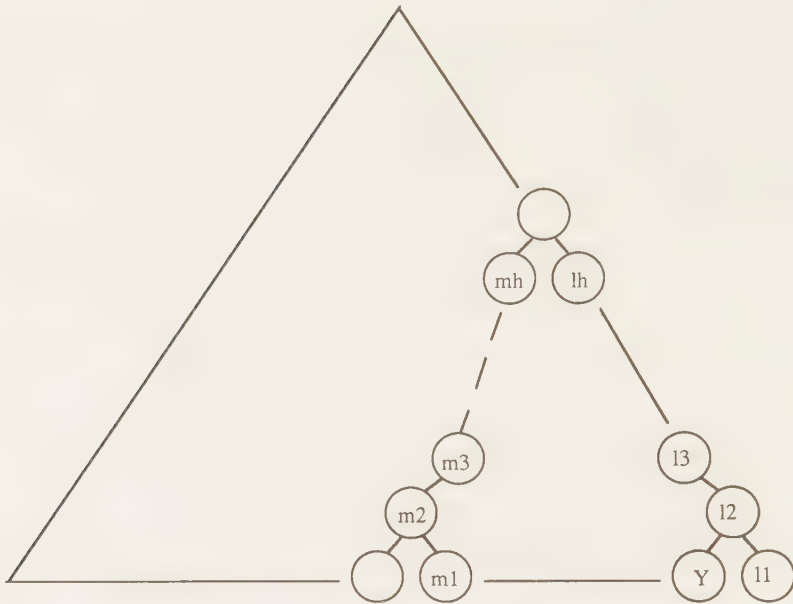


Figure 4.2: Labeling of nodes in a heap with $2^k - 1$ elements.

Since we have two sequences of three ordered elements we can easily get the probability for one element being smaller than an element in the other sequence. We also have the information of which sequence has the largest element. Now we get

$$\left(\frac{9}{10} \cdot \frac{1}{2} + \frac{7}{10} \cdot \frac{1}{2} \right) P(l_2 < m_2) + \left(\frac{7}{10} \cdot \frac{1}{2} + \frac{3}{10} \cdot \frac{1}{2} \right) P(l_2 > m_2) =$$

$$\frac{8}{10} P(l_2 < m_2) + \frac{1}{2} (1 - P(l_2 < m_2)) =$$

$$\frac{1}{2} + \frac{3}{10} P(l_2 < m_2)$$

$P(l_2 < m_2)$ for a fixed h is equal to $P_{1|h-1}$, since we can remove all elements in the bottom row and solve the problem for a heap with one element shorter paths. We also know that $P_{1|1} = 1$, so we get:

$$P_{1|h} = \begin{cases} 1 & \text{if } h=1 \\ \frac{1}{2} + \frac{3}{10} P_{1|h-1} & \text{otherwise} \end{cases}$$

$$P_{1|h} = \frac{1}{2} \sum_{k=0}^{h-2} \left(\frac{3}{10}\right)^k + \left(\frac{3}{10}\right)^{h-1} = \frac{5}{7} + \frac{2}{7} \left(\frac{3}{10}\right)^{h-1}$$

$$P_{2|h} = P(l_1 < m_2) =$$

$$P(l_1 < m_2 \mid l_2 < m_2) P(l_2 < m_2) + P(l_1 < m_2 \mid l_2 > m_2) P(l_2 > m_2) =$$

$$1 \cdot P(l_2 < m_2) + (P(l_1 < m_2 \mid l_2 > m_2 \wedge l_1 < Y) P(l_1 < Y) +$$

$$P(l_1 < m_2 \mid l_2 > m_2 \wedge l_1 > Y) P(l_1 > Y)) P(l_2 > m_2) =$$

$$P(l_2 < m_2) + \left(\left(1 - 1 / \binom{8}{2}\right) \frac{1}{2} + \left(\left(1 - 7 / \binom{8}{2}\right) \frac{1}{2} \right) (1 - P(l_2 < m_2)) \right) =$$

$$\frac{6}{7} + \frac{1}{7} P(l_2 < m_2) = \frac{6}{7} + \frac{1}{7} P_{1|h-1} = \frac{47}{49} + \frac{2}{49} \left(\frac{3}{10}\right)^{h-2}$$

By repeating the technique we get:

$$P_{i|h} = P(l_1 < m_i) =$$

$$P(l_2 < m_i) + (P(l_1 < m_i \mid l_2 > m_i \wedge l_1 < Y) P(l_1 < Y) +$$

$$P(l_1 < m_i \mid l_2 > m_i \wedge l_1 > Y) P(l_1 > Y)) P(l_2 > m_i) =$$

$$P(l_2 < m_i) + \left(\left(1 - 1 / \binom{2^{i+1}}{2}\right) \frac{1}{2} +$$

$$\left(\left(1 - \left(2^{i+1} - 1\right) / \binom{2^{i+1}}{2}\right) \frac{1}{2} \right) (1 - P(l_2 < m_i)) =$$

$$\frac{2^{i+1}-2}{2^{i+1}-1} + \frac{1}{2^{i+1}-1} P(l_2 < m_i)$$

Since $P(l_2 < m_i) = P_{i-1|h-1}$, we get:

$$P_{i|h} = \begin{cases} \frac{5}{7} + \frac{2}{7} \left(\frac{3}{10} \right)^{h-1} & \text{if } i=1 \\ \frac{2^{i+1}-2}{2^{i+1}-1} + \frac{1}{2^{i+1}-1} P_{i-1|h-1} & \text{otherwise.} \end{cases}$$

This gives us:

$$P_{i|h} = 1 - S_i \left(1 - \left(\frac{3}{10} \right)^{h-i} \right), \text{ where } S_i = \frac{6}{7} \prod_{k=1}^i \frac{1}{2^{k+1}-1} \text{ for } i > 0.$$

Let $E[X|h]$ be the expected number of comparisons to insert the last leaf in the path of maximum sons for a fixed h . $P_{0|h}$ is of course 0, so:

$$E[X|h] = \sum_{k=1}^h (1 - P_{k-1|h}).$$

Let $E[X]$ be the expected number of comparisons over all h , and let p_h be the probability that the length of the path of maximum sons that differs from the path from the last leaf to the root is h .

$$E[X] = \sum_{h=1}^{\lfloor \log n \rfloor - 1} p_h E[X|h] = \sum_{h=1}^{\lfloor \log n \rfloor - 1} p_h \sum_{k=1}^h (1 - P_{k-1|h}) =$$

$$\sum_{h=1}^{\lfloor \log n \rfloor - 1} p_h \left(1 + \sum_{k=2}^h \left(1 - \left(1 - S_{k-1} \left(1 - \left(\frac{3}{10} \right)^{h-(i-1)} \right) \right) \right) \right) <$$

$$\sum_{h=1}^{\lfloor \log n \rfloor - 1} p_h \left(1 + \sum_{k=2}^h S_{k-1} \right) \leq 1.329... ,$$

$$\text{since } \sum_{i=1}^{\infty} S_i = 0.329... \text{ and } \sum_{h=1}^{\lfloor \log n \rfloor - 1} p_h = 1.$$

Theorem 4.2: It is sufficient to do $\log(n+1) + 0.329...$ comparisons between elements on the average to delete the maximum element in a random heap of $2^k - 1$ elements.

Proof: To find the path of maximum sons requires one comparison for each node in the path, and since the length of the path is $\log(n+1) - 1$, the theorem follows from Lemma 4.1.

Let us now work under the assumption that the destroyed randomness, due to repeated deletion of the maximum element, does not make later deletions more costly. If this is true we just have to sum the average cost to delete the maximum for all sizes up to n , to get an upper bound on the average cost of HEAPSORT.

Lemma 4.2: For any random heap it holds that $P_{i|h} \geq 1 - S_i \left(1 - \left(\frac{3}{10} \right)^{h-i} \right)$,

$$\text{where } S_i = \frac{6}{7} \prod_{k=1}^i \frac{1}{2^{k+1} - 1} , \text{ and } E[X|h] \leq (1 - P_{k-1|h})$$

Proof: To find the average cost for deleting the maximum element, we have to consider four different cases. (1) The path of maximum sons is of the same length as the path from the last leaf to the root, and the last leaf is a right-child, (2) the path of maximum sons is of the same length as the path from the last leaf to the root, and the last leaf is a left-child, (3) the path of maximum sons is shorter than the path from the last leaf to the root, and the last leaf is a right-child, and (4) the path of maximum sons is shorter than the path from the last leaf to the root, and the last leaf is a left-child.

Case 1: The path of maximum sons is of the same length as the path from the last leaf to the root, and the last leaf is a right-child (Figure 4.3). The analysis for $E[X|h]$ in this case is exactly the same as in Lemma 4.1.

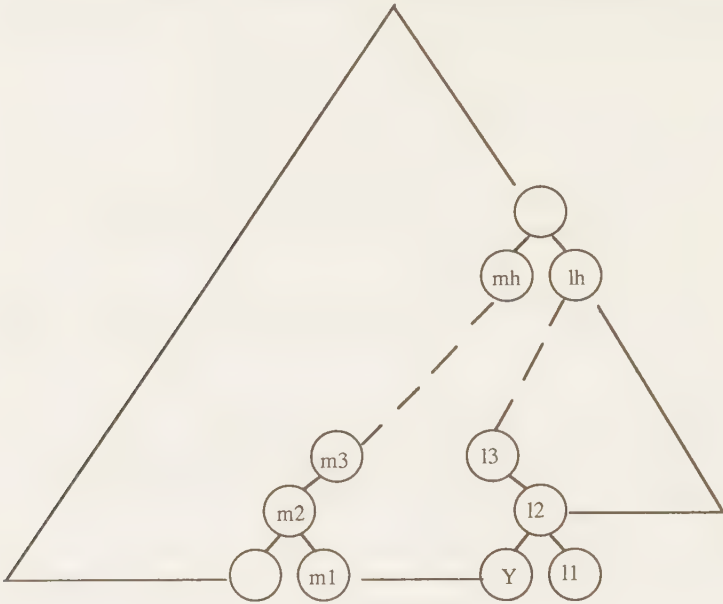


Figure 4.3: A heap where the path of maximum sons is of the same length as the path from the last leaf to the root, and the last leaf is a right-child.

Case 2: The path of maximum sons is of the same length as the path from the last leaf to the root, and the last leaf is a left-child. (Figure 4.4.)

$E[X|h]$ is less than when the last leaf is a right-child. This we get by the same technique as before.

$$\begin{aligned}
 P_{1|h} &= P(l_1 < m_1) = \\
 &P(l_1 < m_1 \mid l_2 < m_2) P(l_2 < m_2) + P(l_1 < m_1 \mid l_2 > m_2) P(l_2 > m_2) = \\
 &\frac{5}{6} P(l_2 < m_2) + \frac{2}{4} (1 - P(l_2 < m_2)) = \frac{1}{2} + \frac{1}{3} P(l_2 < m_2)
 \end{aligned}$$

Since l_2 is the largest of two elements it is clear that its average value is smaller than if it was the largest of three. From this it is easy to see that

$$P(l_2 < m_2) > \frac{5}{7} + \frac{2}{7} \left(\frac{3}{10} \right)^{h-2},$$

and that P_{lh} here is larger than when the last leaf is a right-child. This will also make $E[X|h]$ smaller.

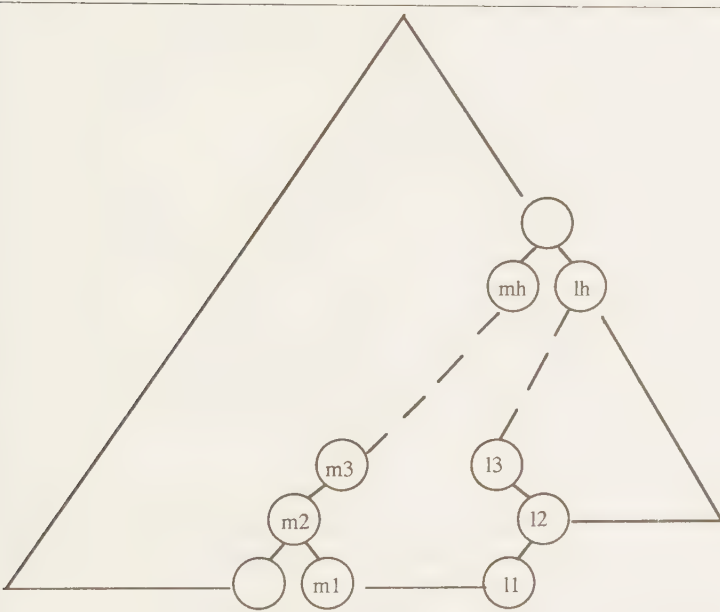


Figure 4.4: A heap where the path of maximum sons is of the same length as the path from the last leaf to the root, and the last leaf is a left-child.

Case 3: The path of maximum sons is shorter than the path from the last leaf to the root, and the last leaf is a right-child. (Figure 4.5.)

If the subheap with the root at l_h has $2^k - 1$ nodes, the analysis is the same as in Lemma 4.1. All l_i that are roots of subheaps with a size less than $2^k - 1$ for any k has a smaller value on the average than if it was of size $2^k - 1$ for the same reasons as in case 2. This will make $E[X[h]]$ smaller.

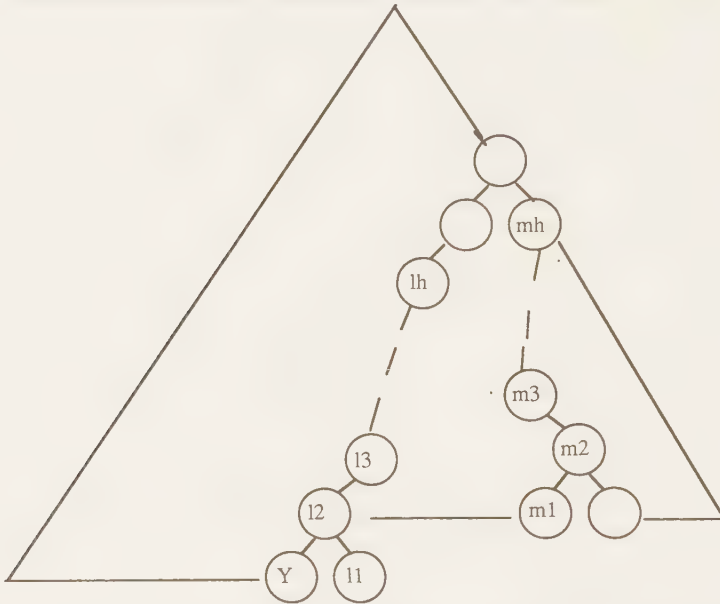


Figure 4.5: A heap where the path of maximum sons is shorter than the path from the last leaf to the root, and the last leaf is a right-child.

Case 4: The path of maximum sons is shorter than the path from the last leaf to the root, and the last leaf is a left-child. (Figure 4.6.)

This is a combination of Cases 2 and 3, so the result follows directly.

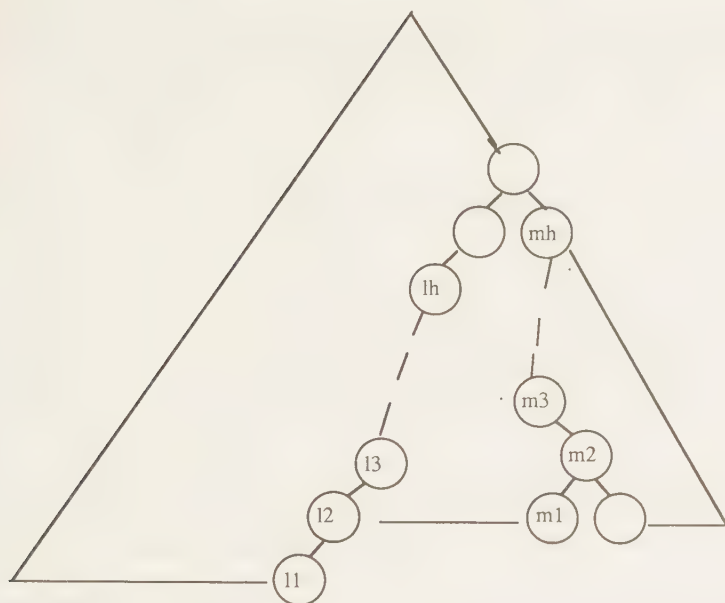


Figure 4.6: A heap where the path of maximum sons is shorter than the path from the last leaf to the root, and the last leaf is a left-child.

Lemma 4.3: The average number of comparisons to insert the last leaf in the path of maximum sons is not more than 1.329 for any random heap.

Proof: It is proved the same way as in Lemma 4.1, using the result of Lemma 4.2.

Theorem 4.3: To delete the maximum element in a random heap $\lfloor \log n \rfloor + 0.329 + M(n)$ comparisons between elements suffices, where

$$M(n) = \begin{cases} 0 & \text{if } n=2 \\ 1 & \text{if } n=3 \\ \left(1 - \frac{2^{\lfloor \log n \rfloor - 1}}{n-1}\right) M(n-2^{\lfloor \log n \rfloor - 1}) & \text{if } 2^k \leq n < 3 \cdot 2^{k-1} \\ \frac{2^{\lfloor \log n \rfloor} - 1}{n-1} + \left(1 - \frac{2^{\lfloor \log n \rfloor} - 1}{n-1}\right) M(n-2^{\lfloor \log n \rfloor}) & \text{otherwise.} \end{cases}$$

Proof:

Due to Lemma 4.3 it is enough to show that the average number of comparisons to find the path of maximum sons is $\lfloor \log n \rfloor - 1 + M(n)$.

Any path in a heap of size n is either of length $\lfloor \log n \rfloor - 1$ or $\lfloor \log n \rfloor$. If $M(n)$ is the probability that it is necessary to do $\lfloor \log n \rfloor$ comparisons to find the path of maximum sons, then the average number of comparisons is $\lfloor \log n \rfloor - 1 + M(n)$.

The probability that the path of maximum sons is found in $\lfloor \log n \rfloor$ comparisons is equal to the probability that the root of the left subheap is larger than the root of the right subheap, multiplied by the probability that the path of maximum sons in the left subheap is found in $\lfloor \log n \rfloor - 1$ comparisons, plus the corresponding probabilities for the right subheap.

Since we have a random heap, the probability for the second largest element being in any subheap is proportional to the size of the subheap. If the last leaf is in the left subheap, i.e. $2^k \leq n < 3 \cdot 2^{k-1}$, the size of the right subheap is $2^{\lfloor \log n \rfloor - 1} - 1$, and the left $n - 2^{\lfloor \log n \rfloor - 1}$. The longest path in the right subheap is $\lfloor \log n \rfloor - 2$, so the probability that we need $\lfloor \log n \rfloor - 1$ comparisons to find the path of maximum sons in the right subheap is zero. This gives us the probability for the path of maximum sons being found in $\lfloor \log n \rfloor$ comparisons as:

$$M(n) = \left(1 - \frac{2^{\lfloor \log n \rfloor - 1}}{n-1} - 1\right) M(n-2^{\lfloor \log n \rfloor - 1}) .$$

If the last leaf is in the right subheap, the probability that $\lfloor \log n \rfloor - 1$ comparisons is necessary to find the path of maximum sons in the left subheap is one. The left subheap has $2^{\lfloor \log n \rfloor} - 1$ elements, and the right $n - 2^{\lfloor \log n \rfloor}$. This gives a total probability of:

$$\frac{2^{\lfloor \log n \rfloor} - 1}{n - 1} + \left(1 - \frac{2^{\lfloor \log n \rfloor} - 1}{n - 1}\right) M(n - 2^{\lfloor \log n \rfloor})$$

that $\lfloor \log n \rfloor$ comparisons are necessary to find the path of maximum sons.

In a heap with only two elements the path of maximum sons is given and no comparison has to be made, but if there are three elements we have to determine which of the two possible paths is the one of maximum sons, using one comparison.

4.3. An Average-Case Analysis of the New HEAPSORT

The HEAPSORT algorithm sorts n elements by first building a heap, and then repeatedly deleting the maximum element, storing it in the "free" position of the array. If the deletion of the maximum element had preserved randomness of the heap, there would not be any problem to find the average number of comparisons for HEAPSORT.

Unfortunately this is not the case, because after a deletion the former last leaf is smaller than all elements in both the path from its present and former location, to the root. This will increase the probability for that element to be and stay among the leaves, and also decrease the average number of comparisons, when the largest element is deleted. This means that a heap will be more "sorted" after some deletions, than a random heap.

The probability for a root of a subheap to be in the path of maximum sons depends on the sizes of the original subheaps, instead of the current sizes, at least until the first former leaf is the root of a subheap, since this is the same as the probability that the second largest element of the remaining was in that subheap at the beginning. This will increase the probability of letting the root of the right subheap be in that path, and thus the probability for having to do $\lfloor \log n \rfloor$ comparisons to find the path is decreased.

According to these arguments the average cost for sorting n elements with the new HEAPSORT algorithm will be less than the sum of deletions of the maximum element for all sizes up to n , plus the cost for building a heap.

Lemma 4.4: The sum of all $M(i)$, for i from 1 to n , is less than $0.721 n$, where $M(n)$ is defined as in Theorem 4.3.

Proof: To prove that the sum of $M(i)$ over all sizes up to n is less than $0.721 n$, it is sufficient to prove that the sum for all heaps of the same height, i.e. all heaps with the same length of the longest path from a leaf to the root, is less than $0.721 \cdot (\text{number of heaps of that height})$.

Suppose it is true for all heaps with the same height less than k . We also know that there are 2^k different sizes of heaps that are of height k , namely 2^k to $2^{k+1} - 1$. Half of them have their last leaf in the left subheap, and the other half in the right. If we sum all $M(i)$ from 2^k to $2^{k+1} - 1$ we get:

$$\begin{aligned}
 & \sum_{i=2^k}^{2^{k+1}-1} M(i) = \\
 & \sum_{i=2^k}^{3 \cdot 2^{k-1}-1} \left(1 - \frac{2^{k-1}-1}{i-1}\right) M(i-2^{k-1}) + \\
 & \sum_{i=3 \cdot 2^{k-1}}^{2^{k+1}-1} \left(\frac{2^k-1}{i-1} + \left(1 - \frac{2^k-1}{i-1}\right) M(i-2^k) \right) = \\
 & \sum_{i=0}^{2^{k-1}-1} \left(1 - \frac{2^{k-1}-1}{2^k-1+i}\right) M(2^{k-1}+i) + \sum_{i=3 \cdot 2^{k-1}}^{2^{k+1}-1} \frac{2^k-1}{i-1} + \\
 & \sum_{i=0}^{2^{k-1}-1} \left(1 - \frac{2^k-1}{2^k+2^{k-1}-1+i}\right) M(2^{k-1}+i) = \\
 & \sum_{i=0}^{2^{k-1}-1} \left(1 - \frac{2^{k-1}-1}{2^k-1+i} + 1 - \frac{2^k-1}{2^k+2^{k-1}-1+i}\right) M(2^{k-1}+i) + \\
 & (2^k-1) \sum_{i=3 \cdot 2^{k-1}}^{2^{k+1}-1} \frac{1}{i-1} <
 \end{aligned}$$

$$\left(2 - \frac{2^{k-1} - 1}{2^k + 2^{k-1} - 2} - \frac{2^k - 1}{2^{k+1} - 2} \right) \sum_{i=0}^{2^{k-1} - 1} M(2^{k-1} + i) + \\ (2^k - 1) \ln \frac{2^{k+1} - 2}{2^k + 2^{k-1} - 2}$$

Since it is easy to show that the proposition is true for all heights up to 6, we get:

$$M(i) < \left(2 - \frac{31}{94} - \frac{1}{2} \right) 0.721 \cdot 2^{k-1} + 2^k \cdot \ln \frac{126}{94} < 0.721 \cdot 2^k.$$

Theorem 4.4: The expected number of comparisons to sort $n=2^k - 1$ elements is not more than $n \log n + 0.700 n$, under the assumption that it is at least as expensive to delete an element from a random heap, as from a heap of the same size formed after some deletions.

Proof: It follows from earlier results in Theorems 4.1 and 4.3, that the number of comparisons in this new HEAPSORT is:

$$1.650 n + \sum_{i=1}^n (\lfloor \log i \rfloor + 0.329 + M(i)) + O(\log n) =$$

$$1.650 n + \sum_{i=1}^n \lfloor \log i \rfloor + 0.329 n + \sum_{i=1}^n M(i) + O(\log n) <$$

$$1.979 n + \sum_{i=1}^{\log(n+1) - 1} (\log(n+1) - i) \frac{n+1}{2^i} + 0.721 n + O(\log n) =$$

$$2.700 n + (n+1) \log(n+1) \sum_{i=1}^{\log(n+1) - 1} \frac{1}{2^i} - (n+1) \sum_{i=1}^{\log(n+1) - 1} \frac{i}{2^i} + O(\log n) =$$

$$2.700 n + n \log n - 2 n + O(\log n) =$$

$$n \log n + 0.700 n + O(\log n)$$

if $n+1$ is a power of 2.

4.4. Analysis of the Floyd - Williams HEAPSORT Algorithm

In Lemma 4.3 it was shown that it is sufficient with 1.329 comparisons to insert the last leaf in the path of maximum sons. This means that the average length between the inserted element and the leaf is 0.329.

Theorem 4.5: The expected number of comparisons to delete the maximum element in a random heap, using the original algorithm of Floyd-Williams is at least $2(\lfloor \log n \rfloor - 1.329 + L(n))$, where

$$L(n) = \begin{cases} 1 & \text{if } n=2 \text{ or } 3 \\ \left(1 - \frac{2^{\lfloor \log n \rfloor - 1} - 1}{n-1}\right) L(n-2^{\lfloor \log n \rfloor}) & \text{if } 2^k \leq n < 3 \cdot 2^{k-1} \\ \frac{2^{\lfloor \log n \rfloor} - 1}{n-1} + \left(1 - \frac{2^{\lfloor \log n \rfloor} - 1}{n-1}\right) L(n-2^{\lfloor \log n \rfloor}) & \text{otherwise.} \end{cases}$$

Proof: $L(n)$ is the probability that the length of the path of maximum sons is $\lfloor \log n \rfloor$, by the same arguments as for $M(n)$ in Theorem 4.3, but with the difference that the length of the path of maximum sons of a heap of size two is equal to one.

Due to Lemma 4.3, we know that the expected length of the path between the root and the inserted element is not shorter than $\lfloor \log n \rfloor - 1 + L(n) - 0.329$. We also know that two comparisons to move one level down in the path with the inserted element is required, and Theorem 4.5 is proved.

By the same argument as for the average cost for HEAPSORT using the new rearrangement algorithm, we can give an lower bound for the regular HEAPSORT algorithm proposed by Floyd. Since the probability that a root of a subheap will be in the path of maximum sons depends only on the size of the original heap, until it is a former leaf, we can assume that the average length of a path of maximum sons is at least $\lfloor \log n \rfloor - 1/2$ over all heaps of height $\lfloor \log n \rfloor$. The expected distance between the inserted former last leaf and the leaf is less than 0.329, as discussed earlier.

Theorem 4.6: It is necessary to do at least $2n \log n - 3.78 n$ comparisons to sort $n=2^k-1$ elements using Floyd's HEAPSORT, if the average path length between the root and the inserted last leaf is at least $\lfloor \log n \rfloor - 1/2 - 0.329$ for a heap of size n formed by repeated deletions.

Proof: The theorem is proved by summation of the cost to build a heap, which is $1.88 n$ [2], and the cost to find the place for the inserted element, which is twice the path length, for every size up to n .

4.5. Average Number of Interchanges

If we modify the new rearrangement algorithm slightly, as described in Theorem 4.1, we can show that the number of comparisons and interchanges are together twice the length of the path of maximum sons plus one. This, together with the fact that the number of interchanges is the same as when using Williams' rearrangement algorithm, we use to prove

Theorem 4.7: The average number of interchanges when deleting the maximum element of a random heap is $\lfloor \log n \rfloor + 2L(n) - M(n) - z(n)$, where $0.671 < z(n) < 1$, $M(n)$ as defined in Theorem 4.3, and $L(n)$ as in Theorem 4.5.

Proof: We know that the average length of the path of maximum sons is $\lfloor \log n \rfloor - 1 + L(n)$, and that the number of comparisons and interchanges together is $2(\lfloor \log n \rfloor - 1 + L(n)) + 1$. The number of comparisons is not more than $\lfloor \log n \rfloor + M(n) + 0.329$ and not less than $\lfloor \log n \rfloor + M(n)$. The number of interchanges can be found by taking the difference between the number of comparisons and the number of comparisons and interchanges together, which is as stated in the theorem. This is the same number of interchanges as in Williams rearrangement algorithm.

We can also estimate the number of interchanges for HEAPSORT, with both the new and Williams rearrangement algorithm, using a technique similar to earlier theorems.

4.6. Simulation Results

In Table 4.1 simulation results for both the new and Floyd-Williams HEAPSORT algorithm are presented. The new algorithm is clearly better than Floyd - Williams and the simulated results point at $n \log n + 0.4 n$ comparisons on the average.

n	Average number of comparisons (new algorithm)	Average number of comparisons (Floyd-Williams algorithm)
10	32.95	38.63
50	292.60	414.73
100	694.35	1 027.69
500	4 645.00	7 426.05
1 000	10 307.25	16 851.81
5 000	63 323.25	107 685.8
10 000	136 664.30	235 371.9
50 000	799 805.38	1 409 794.

Table 4.1:Simulation results for HEAPSORT. The values for the Floyd-Williams algorithm is taken from [9], and the values for the new algorithm are based on 20 simulations for each size.

4.7. Conclusions

In this chapter a sorting algorithm with a very good average-case performance has been presented. It is not more complicated than Floyd-Williams HEAPSORT algorithm [6], but uses only about half the number of comparisons, and the same number of interchanges. Asymptotically it uses only 69% of the number of comparisons QUICKSORT uses on the average, since QUICKSORT takes $1.38 n \log n$ comparisons on the average [12]. Using the result in [9] for the average number of comparisons in MERGESORT we know that the new HEAPSORT algorithm uses less than $2n$ comparisons more but does not use any extra space, which MERGESORT does.

The worst case of this new variant of HEAPSORT is still left to be investigated. One deletion of the maximum element can take $2 \lfloor \log n \rfloor - 1$ comparisons, which is one less than for the worst case of Williams algorithm. If repeated deletions are performed some must be even less expensive, due to the properties of a heap.

The new algorithm will also improve the average-case behavior of all algorithms that uses this form of implicit heaps as priority queues, and half the number of comparisons when deleting a maximum element.

5. Amortized Cost of a Priority Queue Structure

Contents

A new data structure to implement priority queues is presented. It is shown that the amortized number of comparisons for deleting the smallest element from this structure is exactly $\lfloor \log n \rfloor$, where n is the current number of elements in the priority queue. The maximum, as well as the amortized number of comparisons to insert the n :th element is $\lfloor \log \log n \rfloor + 1$. The same result also applies to the number of interchanges when deleting the smallest element from the data structure, while an insertion uses $\lfloor \log n \rfloor$ interchanges.

5.1. A New Data Structure with Good Amortized Behavior

In many heap based algorithms there is an extra cost for balancing the heap after a deletion, to have enough information on the configuration of the heap to be able to insert and delete elements. In the structure presented here, extra space is used to keep this information instead.

As in Williams' algorithm the elements are stored in an array to form a binary tree, where the root of the tree is stored at the first position of the array, and a node at position k has its sons at $2k$ and $2k+1$, and its father at position $k \div 2$. The elements are so ordered that a node has a higher priority than its sons and lower than its father. In this structure there should also be one stack for each level of the heap, and a boolean array marking if the corresponding position of the heap is empty or not. For simplicity we will regard it as a min-heap.

When the element with highest priority is deleted, i.e. the element stored in the root, it is replaced by its smallest son. Since this son is the root of a subheap we can proceed in the same way recursively until we reach a leaf. This will create a "hole" in the heap, and the address of the hole is pushed onto the stack for that level of the heap.

If an element is to be inserted it should be inserted in a free position as close to the root as possible, and swapped with its father until it is either the root or has lower priority than its father. This position we can find by taking the top element of the non-empty stack closest to the root. Since all stacks further from the root are non-empty we can keep the address of the stack closest to the root in memory and update if necessary in constant time. We can also find that stack in constant time. If all stacks are empty when an element is inserted in a priority queue of n elements, the new element can be inserted at position $n+1$.

If we want to construct a heap of n elements we can use the algorithm of Gonnet and Munro [10] or the one in Chapter 4.

More detailed algorithms are presented in Figure 5.1, where the element with the smallest value has highest priority. It is not very difficult to transform the algorithms to work with pointers instead.

5.2. Analysis of the Amortized Cost

In [14] Tarjan presented a new way of measuring the cost of algorithms, called the amortized cost. The main idea is that, if we do operations that are expensive in the worst case, parts of this may be paid for by less expensive operations performed earlier. In this case it means that the worst number of comparisons when an element is deleted, which is $O(n)$, has been paid for by earlier deletions which has not used $\lfloor \log n \rfloor$ comparisons, so that the average cost over all deletions is $\leq \lfloor \log n \rfloor$.

Theorem 5.1: The amortized number of comparisons for deleting the smallest element from the structure presented above is $\lfloor \log n \rfloor$.

Proof: We would like to find a potential function Φ , which will measure how many comparisons we have saved during cheap deletions, to help paying for an expensive one. This we do if we choose Φ to be the sum of the distances to the root over all nodes minus the sum of $\lfloor \log i \rfloor$ for all i up to the current size of the priority queue.

The amortized cost for an operation is equal to the real cost plus the change of potential of the structure. The change of potential for a delete-min operation is $\lfloor \log n \rfloor$ minus the distance from the new "hole" to the root, and the real number of comparisons is also the distance from the new "hole" to the root. This gives an amortized cost for the delete operation of exactly $\lfloor \log n \rfloor$.

Theorem 5.2: The amortized number of comparisons for inserting an element in the structure presented above is $\lfloor \log \lfloor \log n \rfloor \rfloor + 1$.

Proof: This is proved in the same way as for the delete-min operation, where the change of potential is the distance to the root from the hole now filled minus $\lfloor \log n \rfloor$. Since it is possible to perform a binary search in the path we can add the real cost to a total cost of:

$$\lfloor \log(\text{distance to the root}) \rfloor + 1 + (\text{distance to the root} - \lfloor \log n \rfloor)$$

Since we insert elements in the heap as close to the root as possible, the distance to the root must be less than or equal to $\lfloor \log n \rfloor$, and the amortized cost less than or equal to $\lfloor \log \lfloor \log n \rfloor \rfloor + 1$.

We can use the same technique to get the amortized number of interchanges for both operations, using the same potential function, with the cost of inserting an element equal to the distance to the root, which is less than or equal to $\lfloor \log n \rfloor$.

The extra storage required by this structure, is for the stacks and possibly also for marking nodes as empty. If we cannot allocate space dynamically we need $\lfloor \log m \rfloor$ pointers to the stack tops, an array with m pointers to implement the stacks if the freed space in the heap cannot be used as pointers, and m bits to mark the elements empty, where m is the maximal size of the priority queue. If we have a pointer implementation of the heap we don't need extra space to mark elements empty, and we can allocate place in the stacks when needed.

5.3. Comments

By a fairly simple data structure we improve the best number of comparisons for a priority queue, in the amortized sense, of the best algorithms known. It is also possible to use both a "semi-implicit" and a pointer implementation of the structure, which the implementations in Chapter 3 and Gonnet and Munro can not [10].

In [7] Fredman and Tarjan gives the asymptotic amortized cost for Fibonacci - heaps, but no constants are given. In their structure, however, this is not so important as the fact that both insertions and decreasing the value of a key can be done in constant amortized time.

```

procedure DeleteMin;
var I, level : integer;
begin
    if NrOfElements < 1 then
        ERROR
    else
        begin
            I:= 2; level:=0;
            while I < maxsize and not (Empty [I] and Empty [I + 1]) do
                begin
                    if Empty [ I ] then
                        I:= I + 1
                    else
                        if not Empty [ I + 1] then
                            if A[ I + 1 ] > A[ I ] then
                                I:= I + 1;

                                A[ I div 2 ]:= A[ I ];
                                I:= 2*I;
                                level:= level + 1
                            end;

                            I:= I div 2;
                            Empty[ I ]:= true;
                            PUSH( STACK[ level ] , I );
                            If level < FirstStack then
                                FirstStack:= level;
                            NrOfElements:= NrOfElements - 1
                        end
                    end;
                end;
            end;
        end;
    end;

```



```

procedure Insert( X : elementtype );
var I : integer;
begin
  if NrOfElements = maxsize then
    ERROR
  else
    begin
      NrOfElements:= NrOfElements + 1;
      if StackEmpty( STACK[ FirstStack ] ) then
        FirstStack:= FirstStack + 1;
      if StackEmpty( STACK[ FirstStack ] ) then
        I:= NrOfElements
      else
        I:= POP( STACK[ FirstStack ] );

      while I > 1 do
        if A[ I ] < A[ I div 2 ] then
          begin
            SWAP( A[ I ], A[ I div 2 ] );
            I:= I div 2;
          end
        else
          I:= 0
        end
      end
    end
  end;

```

Figure 5.1: The algorithms for deletion and insertion in the new data structure, where A is the array storing the heap, STACK is an array of stacks, and Empty a Boolean array.

If we want to, we can also use our structure for sorting. If we use the algorithm for building a heap presented by Floyd [6] or, even better, the one by Gonnet and Munro [10], before the repeated deletions we get a sorting algorithm which is very close to optimal. The total cost to sort $n=2^k-1$ elements will be:

$$1.625 n + O(\log n) + \sum_{i=1}^n \lfloor \log i \rfloor = n \log n - 0.375 n + O(\log n).$$

If we want to use as little extra space as possible for sorting we don't have to stack the positions of the holes, but just mark them empty. This will take only n extra bits. Gonnet and Munro's algorithm for building a heap requires extra space, so we can use the technique presented in Chapter 4 to get a worst case of $n \log n + O(\log n)$ comparisons, while the average number of comparisons is $n \log n - 0.350 + O(\log n)$.

6. The Deap — a Double-Ended Heap

Contents

This Chapter presents a symmetric implicit double-ended priority queue implementation, which can be built in linear time. The smallest and the largest element can be found in constant time, and deleted in logarithmic time. This structure is an improvement of the MinMaxHeap presented in [1].

6.1. Introduction to Priority Deques

A priority queue is a data type where the element with the smallest key value can be found and deleted, and new elements can be inserted. It is also called a priority queue when the element with the largest key value is wanted, instead of the smallest.

In some cases it is interesting to find and delete both the smallest and the largest element, as well as being able to insert new elements. A structure with these operations may be called a double-ended priority queue, or priority deque for short.

The most common way of implementing priority queues is through a heap. Some attempts to implement double-ended priority queues with heaps or heaplike structures have been made. Some examples of this are the MinMaxHeap [1] and the structure with two heaps placed "back to back" in a suitable way, proposed by Williams.

In this Chapter, a double-ended heap called Deap, is presented. This structure is a mixture of the MinMaxHeap and the priority deque proposed by Williams. Also the use of binary search of a path in a heap is used. This binary search technique is presented in Chapter 3.

The asymptotic results of all three implementations are the same. That is:

1. finding the largest and the smallest element takes constant time,
2. deleting the maximum or minimum element, or inserting a new element takes $O(\log n)$ time,
3. the structure can be constructed in linear time, and
4. no additional pointers are required.

6.2. The Data Structure

A Deap is an implicit data structure much resembling a heap. It can be regarded as a heap with an element at the root that is never referred. The left subtree of the root is a minheap, a heap with the smallest element at the root, and the right is a maxheap, a heap with the largest element at the root. Any leaf in the minheap is also smaller than the corresponding leaf in the maxheap, i. e. any node k in the minheap is smaller than the element at node $k + a$, if it exist, and $\lfloor (k + a)/2 \rfloor$, where a is $2^{\lfloor \log k \rfloor - 1}$. Otherwise the Deap has the same properties as the common heap. The Hasse diagram for a Deap is retrieved by turning the maxheap "up-side-down" and placing it back-to-back with the minheap. An example of a Deap is shown in Figure 6.1, and the corresponding Hasse diagram in Figure 6.2.

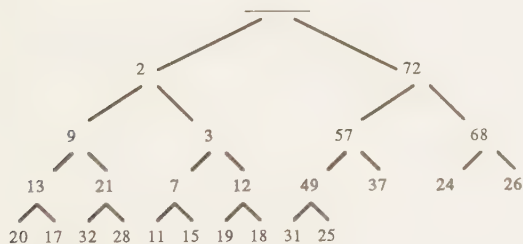


Figure 6.1 Sample of a Deap

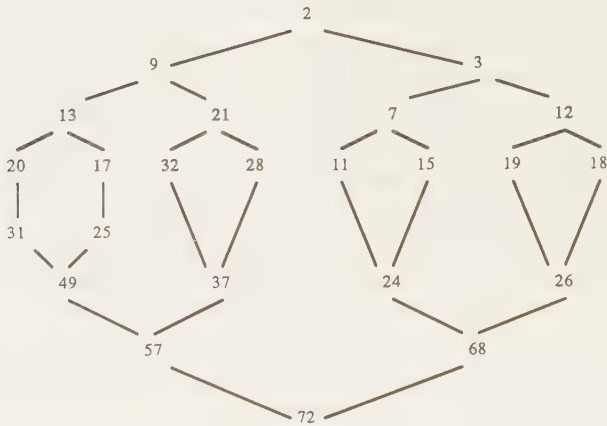


Figure 6.2 Hasse Diagram for the Deap of Figure 6.1

As can be observed in the Hasse diagram, the Deap is a symmetric structure, which makes it easy to transform so that the left subtree is a minheap and the right is a maxheap, without using any extra space.

6.3. Algorithms for the Operations

Since the minimum element in a Deap is stored in position two and the maximum in three it is easy to find them in constant time.

If an element is to be inserted in the minheap, it has to be compared with the corresponding leaf in the maxheap. If the new element is the larger, the two has to change places, and the new element is inserted in the maxheap with the binary insertion technique described in Chapter 3, which is easy to do after observing that a node at position k has its ancestor i levels up at position $k \div 2^i$. If the new element is smaller than its corresponding leaf in the maxheap, it is inserted in the minheap by the same technique. If an element is to be inserted in the maxheap a similar algorithm is used.

If the minimum element has to be deleted, we have to replace it with its smallest son. This son has to be replaced by its smallest son and so on, until we reach a leaf. Then the element stored in the last place of the Deap is inserted in the empty space. The maximum element is deleted in a similar way.

We construct a Deap in the same way as we construct a heap, by repeated delete operations, but instead of inserting the element in the last place, we insert the root of the subDeap we are currently working on.

For further implementation details see Appendix F.

6.4. Complexity Analysis

Theorem 6.1: An element can be inserted in a Deap with at most $\log \log n + 2$ comparisons.

Proof: The height of a heap is $\lfloor \log n \div i \rfloor$, where i is the index of the root and n the index of the last leaf. A binary insertion in a sorted sequence cost $\lfloor \log \text{the length} \rfloor + 1$ comparisons, so after have used one comparison to determine if the new element has to be inserted in the max-heap or the min-heap we get a total number of comparisons of $1 + \lfloor \log \lfloor \log n \div 2 \rfloor \rfloor + 1$.

Theorem 6.2: Both the minimum and the maximum element can be deleted in $\log n + \log \log n$ comparisons.

Proof: The cost of finding the path of minimum, or maximum, sons is equal to the height of the heap, which is at most $\lfloor \log((n+1) \div 2) \rfloor$ and then the binary insertion cost not more than $\lfloor \log \lfloor \log((n+1) \div 2) \rfloor \rfloor + 1$ comparisons.

Theorem 6.3: A Deap can be constructed with $2.07 \dots n$ comparisons.

Proof: Since the creation of a Deap can be seen as repeated deletions we get the following sum:

$$\sum_{i=1}^{3n/4} (\lfloor \log((n+1) \text{ div } i) \rfloor + \lfloor \log \lfloor \log((n+1) \text{ div } i) \rfloor + 1 \rfloor + 1),$$

which adds up to $2.07 \dots n$ comparisons.

These results are to be compared with those for the MinMaxHeap [1], which are for:

Insert: $\log(\log(n+1))$

DeleteMin: $(3/2)\log(n) + \log(\log(n))$

DeleteMax: $(3/2)\log(n) + \log(\log(n))$

Create: $2.15 \dots n$

6.5. Concluding Remarks

Algorithms and data structures for double-ended priority queues have been presented earlier, but none of them have had the symmetry as well as the simple storage requirement of the Deap. In addition to this, it uses fewer comparisons than any other data structure for this purpose. The symmetry should also simplify the merging of priority dequeues, a problem under investigation.

7. Conclusions and Further Research

The main results of this thesis concern the rearrangement procedure in both the worst and the average case. This will effect the delete operation as well as the creation of a heap. Since these are the main parts of HEAPSORT it will have an effect on the sorting time. Some of these results presented are only marginal, while others should be of practical interest.

The use of ternary heaps as described in Chapter 2 is not very useful when paths of maximum sons are found, since it will make it much more expensive to find that path. The heap with scattered leaves, however, improves all the results in both the worst and the average case, but it has to be more investigated for the average case. The reason for its good behavior is that it balances the heap in such a way that the sizes of the subheaps of an element differ by at most one. Unfortunately it is a little more complicated to compute the addresses in such a heap.

In Chapter 3 there are various ways of getting a good worst case behavior, where the first algorithm described is the most useful in practice. The others are more complicated and the small improvement is only notable for very large heaps. The computing time for the binary search in the algorithms is often very long but that depends mostly on bad implementation of the div operation of powers of two. This can easily be implemented by shift operations instead making it fast.

In Chapter 4 the best algorithm of the thesis is presented. It is simple — almost simpler than the original HEAPSORT algorithm by Williams [15] — and it is very fast on the average. It is just slightly slower than MERGESORT which is the fastest sorting algorithm, but it uses extra space which HEAPSORT does not. Furthermore it will not increase the number of data movements in HEAPSORT much — approximately $0.329 n$ data movements extra. The worst case of the algorithm has to be examined in future work, but it is clear that it is better than for Williams algorithm. I can see no reason why this algorithm should not be presented in textbooks as the HEAPSORT algorithm since it is almost twice as fast as the old algorithm.

The average case analysis of the algorithm in Chapter 4 is also of great interest, since the same technique can be used to analyse the average case of Williams algorithm. There is no analysis made before on the average behavior of HEAPSORT that are very close to the real values, though Doberkat [3] showed that a delete operation from a random heap is asymptotically $2 \log n$. It may be interesting to analyse the corresponding algorithm for heaps with scattered leaves. Simulation results shows an even better behavior on the average, and the analysis there seems to be simpler.

In Chapter 5 a data structure with a good amortized behavior is presented. The most interesting with this is maybe the exact analysis of the amortized number of comparisons. Most analysis of amortized cost are made in the asymptotic sence, which of course is also very useful. There is lots to do in constructing and analysing algorithms which are suitable when amortized cost is a relevant measure.

The double-ended heap in Chapter 6 is just an application of the ideas in Chapter 3, together with some other experiences. What can be done with this is to construct algorithms which works well on the average using the ideas of Chapter 4, and analyse them. It is also interesting to see whether this structure is useful when double-ended priority queues should be merged.

In future work on heaps a parallel algorithm based on HEAPSORT with almost optimal number of comparisons with a small number of processes will be investigated. This idea comes from an attempt to classify and to find common structures in different sorting methods, but this work seems also very promising, where many sorting methods can be regarded as variants of one or maybe two super algorithms.

References

1. Atkinson M.D.; Sack J.-R.; Santoro N. and Strothotte Th.; An Efficient, Implicit Double-Ended Priority Queue; Report SCS-TR-55 (july 1984)
2. Doberkat E.E; An Average Case Analysis of Floyd's Algorithm to Construct Heaps; Information and Control, 61: 114- 131 (1984)
3. Doberkat E.E; Deleting the Root of a Heap; Acta Informatica, 17: 245- 265 (1982)
4. Doberkat E.E; Inserting a New Element into a Heap; BIT, 21: 255- 269 (1981)
5. Erkiö H.; On HEAPSORT and its Dependence on Input Data; Report A- 1979- 1, Department of Computer Science, University of Helsinki (1979)
6. Floyd R.W.; Algorithm 245, Treesort; CACM, 7(12): 701 (dec 1964)
7. Fredman L. and Tarjan R.E.; Fibonacci Heaps and their uses in Improved Network Optimization Algorithms; Proceedings 25th Annual IEEE Symposium on foundations of Computer Science (1984)
8. Göbel F. and Hoede C.; On an Optimality Property of Ternary Trees; Information and Control 42: 10- 26 (1979)
9. Gonnet G.H.; Handbook of Algorithms and Data Structures; Addison- Wesley (1984)
10. Gonnet G.H. and Munro J.I.; Heaps on Heaps; Proceedings ICALP, Aarhus 9:282- 291 (July 12- 16 1982)
11. Knuth D.E.; The Art of Computer Programming, vol III: Sorting and Searching; Addison- Wesley, Reading Mass. (1973)
12. Mehlhorn K.; Data Structures and Algorithms 1: Sorting and Searching; Springer- Verlag (1984)

13. Porter Th. and Simon I.; Random Insertion into a Priority Queue Structure; IEEE Trans Software Engineering, 1 SE: 292- 298 (1975)
14. Tarjan R.E.; Amortized Computational Complexity; SIAM J. Alg. Disc. Meth. 6: 306-318 (1985)
15. Williams J.W.J.; Algorithm 232; CACM, 7(6): 347- 348 (June 1964)

APPENDICES

Appendix A

The following declaration is in effect for all of the algorithms for HEAPSORT, given below

```
Type vector = array [ 1..N ] of elementtype;
```

where N is the number of elements to be sorted, and elementtype is the type of element to be sorted.

A.1. The Algorithm for HEAPSORT Using a Complete Binary Heap

```
procedure HEAPSORT (var A : vector; N : integer);
  (* N is the number of elements to be sorted *)

var I      : integer;
      TEMP : elementtype;

begin
  for I:= N div 2 downto 1 do
    REARRANGE(A,I,N);
    (* A heap is created *)

  for I:= N-1 downto 1 do
    begin
      TEMP:=A[1]; A[1]:=A[I+1]; A[I+1]:=TEMP;
      (* The largest element of the remaining changes
         place with the last of the remaining *)
      REARRANGE(A,1,I)
    end
  end (* of HEAPSORT *) ;
```

```

procedure REARRANGE (var A : vector; I,N : integer);

var J : integer;
    X : elementtype;

begin
    X:=A[I];
    J:=2*I;  (* A[J] = the left son of A[I] *)

    while J <= N do
    begin
        if J+1 <= N then
            if A[J] <= A[J+1] then
                J:=J+1;  (* A[J] = the largest son of A[I] *)
            if X < A[J] then
                begin
                    A[I]:=A[J];
                    I:=J;
                    J:=2*J;
                end
            else
                J:=N+1; (* Exit the while loop *)
            end;
        A[I]:=X
    end (* of REARRANGE *) ;

```

A.2 The Algorithm for HEAPSORT Using a Complete Ternary Heap

```

procedure TREAPSORT (var A : vector; N : integer);

var I,TEMP : integer;

begin
    for I:= (N-1) div 3 downto 1 do
        REARRANGE(A,I,N);

    for I:= N-1 downto 1 do
        begin
            TEMP:=A[1]; A[1]:=A[I+1]; A[I+1]:=TEMP;
            REARRANGE(A,1,I)
        end
    end (* of TREAPSORT *) ;

```

```

procedure REARRANGE (var A :vector; I,N :integer);

var J,TEMP : integer;
    X       : elementtype;

begin
  X:=A[I];
  J:=3*I-1;
  while J <= N do
    begin
      TEMP:=J;
      if J+1 <= N then
        begin
          if A[J+1] >= A[J] then
            TEMP:=J+1;
          if J+2 <= N then
            begin
              if A[J+2] >= A[TEMP] then
                TEMP:=J+2;
            end
          else
            J:=N+1;
          end
        end
      else
        J:=N+1;
      end
      if X < A[TEMP] then
        begin
          A[I]:=A[TEMP];
          I:=TEMP;
          J:=3*I-1;
        end
      else
        J:=N+1;
      end;
      A[I]:=X
    end (* of REARRANGE *);

```

A.3 The Algorithm for HEAPSORT Using a Binary Heap with Scattered Leaves

```

procedure HEAPSORT (var A : vector; N : integer);

var I,J,K,START : integer;
    TEMP          : elementtype;

begin
    K:=round( exp( ln(2)*trunc( ln(N)/ln(2) ) ) );
    J:=K;
    if N < 3 * (K div 2) then
        START:=N - (K div 2)
    else
        START:=K-1;

    for I:= START downto 1 do
        begin
            if I < K then K:=K div 2;
            REARRANGE(A, I, N, K);
        end;

    for I:= N-1 downto 1 do
        begin
            TEMP:=A[I+1]; A[I+1]:=A[1];
            if I+1 < J then J:=J div 2;
            XREARRANGE(A, I+1, I+1-J, TEMP)
        end
    end (* of HEAPSORT *);

```

```

procedure XREARRANGE (var A : vector; N,N1 : integer; X : elementtype);

var J,K,L1,M,Acc,Lev : integer;
    READY                : boolean;

begin
    M:=2; Acc:=1; Lev:=2; K:=1;
    READY:= false;

    repeat
        J:=N1 mod Lev;
        L1:=(J+Acc) mod Lev;
        if A[M+J] >= A[M+L1] then
            begin
                A[K]:=A[M+J];
                K:=M+J;
                M:=M+Lev;
                Acc:=Lev;
                Lev:=Lev*2
            end
        else
            begin
                A[K]:=A[M+L1];
                A[M+L1]:=X;
                READY:=true;
                REARRANGE(A,M+L1,N-1,Lev)
            end
    until (K+Acc*2 > N-1) or READY;

    if not READY then
        if (K+Acc < N) and (A[K+Acc] > X) then
            begin
                A[K]:=A[K+Acc];
                A[K+Acc]:=X
            end
        else
            A[K]:=X
        end
    end (* of XREARRANGE *);

```



```

procedure REARRANGE (var A : vector; I,N,Acc :integer);

var J : integer;
    X : elementtype;

begin
  X:=A[I];
  J:= I+Acc;

  while J <= N do
    begin
      if J+Acc <= N then
        if A[J+Acc] >= A[J] then
          J:=J+Acc;
        if X <A[J] then
          begin
            A[I]:=A[J];
            I:=J;
            Acc:=Acc*2;
            J:=J+Acc
          end
        else
          J:=N+1
        end;
      A[I]:=X
    end (* of REARRANGE *);

```

If the call for XREARRANGE in HEAPSORT is changed to REARRANGE(A,1,I,J) the algorithm will perform its task without calling on XREARRANGE, but after some extra comparisons between elements.

A.4 Algorithm for HEAPSORT Using a Ternary Heap with Scattered Leaves

```

procedure TREAPSORT (var A : vector; N : integer);

var I,J,K,START : integer;
    TEMP          : elementtype;

begin
    K:=round( exp( ln(3)*trunc( ln(2*N)/ln(3) ) ) );
    J:=K;
    if 2*N < ( 5*K div 3 ) then
        START:=N-K div 3
    else
        START:=K div 2;

    for I:= START downto 1 do
        begin
            if 2*I < K then K:=K div 3;
            REARRANGE(A,I,N,K)
        end;

        for I:=N-1 downto 1 do
            begin
                TEMP:=A[I+1]; A[I+1]:=A[1];
                if 2*(I+1) < J then J:=J div 3;
                XREARRANGE(A,I+1,I+1-((J-1) div 2 + 1),TEMP)
            end
        end (* of TREAPSORT *);

procedure XREARRANGE (var A : vector;N,N1: integer;X:elementtype);

var J,K,L1,L2,M,Acc,Lev,TEMP : integer;
    READY
        : boolean;

begin
    M:=2; Acc:=1; Lev:=3; K:=1;
    READY:=false;
    while ( K+3*Acc < N ) and not READY do
        begin
            J:=N1 mod Lev;
            L1:=(J+Acc) mod Lev;
            L2:=(L1+Acc) mod Lev;
            if A[M+J] >= A[M+L1] then
                if A[M+J] >= A[M+L2] then
                    begin
                        A[K]:=A[M+J];
                        K:=M+J;
                        M:=M+Lev;
                        Acc:=Lev;
                        Lev:=Lev*2;
                    end
                end
            end
        end

```

```

else
begin
  A[K] := A[M+L2];
  A[M+L2] := X;
  READY := true;
  REARRANGE (A, M+L2, N, Lev)
end
else
if A[M+L1] >= A[M+L2] then
begin
  A[K] := A[M+L1];
  A[M+L1] := X;
  READY := true;
  REARRANGE (A, M+L1, N, Lev)
end
else
begin
  A[K] := A[M+L2];
  A[M+L2] := X;
  READY := true;
  REARRANGE (A, M+L2, N, Lev)
end
end;

if not READY then
begin
  if K+Acc < N then
  begin
    TEMP := K+Acc;
    if K+2*Acc < N then
      if A[K+2*Acc] > A[TEMP] then
        TEMP := K+2*Acc;
      if A[TEMP] > X then
        begin
          A[K] := A[TEMP];
          A[TEMP] := X;
        end
      else
        A[K] := X
      end
    end
  end
  A[K] := X;
end
end (* of XREARRANGE *);

```

```
procedure REARRANGE (var A : vector; I,N,Acc : integer);  
  
var J,TEMP : integer;  
    X      : elementtype;  
  
begin  
    X:=A[I];  
    J:=I+Acc;  
  
    while J <= N do  
    begin  
        TEMP:=J;  
        if J+Acc <= N then  
        begin  
            if A[J+Acc] >= A[J] then  
                TEMP:=J+Acc;  
            if J+2*Acc <= N then  
            begin  
                if A[J+2*Acc] >= A[TEMP] then  
                    TEMP:=J+2*Acc  
            end  
            else  
                J:=N+1  
            end  
        else  
            J:=N+1;  
        if X < A[TEMP] then  
        begin  
            A[I]:=A[TEMP];  
            I:=TEMP;  
            Acc:=Acc*3;  
            J:=I+Acc  
        end  
        else  
            J:=N+1  
        end;  
        A[I]:=X  
    end (* of REARRANGE *);
```

The same changes as described in Appendix A.3 have the same effect as described there.

Appendix B

The same declaration as in Appendix A is in effect for the algorithms in Appendix B.

B.1 The Algorithm for Inserting an Element in a Complete Binary Heap

```

procedure INSERTINHEAP ( var A : vector; X : elementtype; N :integer);
(* Insert element X in a heap stored in A with N-1 elements *)

var TOP,BOT,MID,I : integer;

begin
  BOT:= trunc ( ln(N)/ln(2) ) + 1;
  TOP:= 1;
  while BOT > TOP do
    begin
      MID:=( TOP + BOT ) div 2;
      if X <= A[N div round(exp(ln(2)*MID))] then
        BOT:=MID;
      else
        TOP:=MID+1
      end;

    for I:= 1 to BOT - 1 do
      begin
        A[N]:=A[N div 2];
        N:=N div 2
      end;
      A[N]:=X
    end;

```

B.2 The Algorithm for Inserting an Element in a Ternary Heap with Scattered Leaves

```

procedure INSERTINHEAP(var A : vector; X : elementtype; N : integer);

var N1, TOP, , BOT, MID, MIDACC, I : integer;

begin
  DEPTH:= trunc( ln(2*n) / ln(3));
  LEFTMOST:= round( exp( ln(3)*DEPTH ) div 2 + 1;
  N1:=N - LEFTMOST;

  TOP:=0;
  BOT:=DEPTH;

  while BOT > TOP do
    begin
      MID:= (TOP + BOT ) div 2
      MIDACC:= round( exp( ln(3)*MID ));
      if X > A[ N1 mod MIDACC + MIDACC div 2 +1 ] then
        BOT:= MID
      else
        TOP:= MID + 1
      end;

  DEPTHACC:= round( exp( ln(3)*DEPTH ));

  for I:= DEPTH downto TOP + 1 do
    begin
      A[ LEFTMOST + N1 mod DEPTHACC ]:=
        A[ LEFTMOST - (DEPTHACC div 3) + N1 mod (DEPTHACC div 3) ];
      DEPTHACC:= DEPTHACC div 3;
      LEFTMOST:= LEFTMOST - DEPTHACC
    end;
    A[ LEFTMOST + N1 mod DEPTHACC ]:= X
  end;

```

Appendix C

C.1. A HEAPSORT with Good Worst-Case Behavior

```

procedure REARRANGE(var A : vector; X : elementtype; I,N : integer);

var J, K, NrOfLevels,
    Bot, Top, Mid : integer;

begin
    J := 2*I;

    { *** LINEAR SEARCH *** }
    while J < N do
        if A[ J ] < A[ J+1 ] then
            J := 2*(J+1)
        else
            J := 2*J;

    if J > N then
        J := J div 2;           { J is now a leaf }

    { *** BINARY SEARCH *** }

    NrOfLevels := log( J div I );
    { Number of levels between I and J }
    TOP := NrOfLevels;
    BOT := 0;

    while Top > Bot do
        begin
            Mid := (Top+Bot) div 2;
            if X ≤ A[ J div 2**Mid ] then
                Top := Mid
            else
                Bot := Mid+1
        end;

    { *** INSERTION *** }

    for K := NrOfLevels - 1 downto Top do
        A[ J div 2**(K+1) ] := A[ J div 2**K ];

    A[ J div 2**Top ] := X;

end;

```

```

procedure HEAPSORT(var A : vector; N : integer);
{ *** This procedure is similar to the usual HEAPSORT *** }

var I      : integer;
      Temp : elementtype;

begin

{ *** HEAP CREATION *** }

  for I:= N div 2 downto 1 do
    REARRANGE(A,A(I),I,N);

{ *** SORTING *** }

  for I:= N-1 downto 1 do
    begin
      Temp:= A[ I+1 ]; A[ I+1 ]:= A[ 1 ];
      REARRANGE(A,Temp,1,I)
    end;

end;

```

The HEAPSORT algorithm implemented in PASCAL, where $2^{**}k$ means 2^k , and $\log(n)$ is a function that returns $\lfloor \log_2 n \rfloor$.

Appendix D

D.1. A Rearrangement Algorithm for Heaps with a Good Average Case Behavior

```

procedure rearrange( var A : vector; I,N : integer; X : elementtype),
var J   :   integer;
begin
    { find path of maximum sons }

    J:= 2*I;

    while J < N do
    begin
        if A[ J+1 ] >= A[ J ] then
            J:= J+1;

            A[ J div 2 ]:= A[ J ];
            J:= 2*J
        end;

        if J > N then
            J:= J div 2
        else
            A[ J div 2 ]:= A[ J ];

            { insert in path of maximum sons }

            while J > I do
                if X > A[ J div 2 ] then
                    begin
                        A[ J ]:= A[ J div 2 ];
                        J:= J div 2;
                    end
                else
                    I:= J;
                end

            A[ J ]:= X

        end;

```

A is the array storing the heap, I is the first and N the last element involved in the rearrangement, and X is the element to be inserted in the path of maximum sons.

Appendix E

E.1. Insertion and Delete-Min Algorithms for a Heap with Good Amortized Behavior

The algorithms for deletion and insertion in a new data structure with good amortized behavior, where A is the array storing the heap, STACK is an array of stacks, and Empty a boolean array.

```

procedure DeleteMin;
var I, level : integer;
begin
  if NrOfElements < 1 then
    ERROR
  else
    begin
      I := 2; level := 0;
      while I < maxsize and not (Empty [I] and Empty [I + 1]) do
        begin
          if Empty [ I ] then
            I := I + 1
          else
            if not Empty [ I + 1] then
              if A[ I + 1 ] > A[ I ] then
                I := I + 1;

            A[ I div 2 ] := A[ I ];
            I := 2*I;
            level := level + 1
          end;

          I := I div 2;
          Empty[ I ] := true;
          PUSH( STACK[ level ] , I );
          if level < FirstStack then
            FirstStack := level;
          NrOfElements := NrOfElements - 1
        end
      end;
    end;
  end;

```

```
procedure Insert( X : elementtype );
var I : integer;
begin
  if NrOfElements = maxsize then
    ERROR
  else
    begin
      NrOfElements:= NrOfElements + 1;
      if StackEmpty( STACK[ FirstStack ] ) then
        FirstStack:= FirstStack + 1;
      if StackEmpty( STACK[ FirstStack ] ) then
        I:= NrOfElements
      else
        I:= POP( STACK[ FirstStack ] );

      while I > 1 do
        if A[ I ] < A[ I div 2 ] then
          begin
            SWAP( A[ I ], A[ I div 2 ] );
            I:= I div 2;
          end
        else
          I:= 0
        end
      end
    end
end;
```

Appendix F

F.1. Implementation of the Operations on a Deap

If the n elements in the Deap is stored in an array at positions 2 through $n+1$ the operations of the Deap can be implemented as follows:

```

function FindMin
  RETURN A[ 2 ]

function FindMax
  RETURN A[ 3 ]

procedure Insert(x)
  size:=size+1
  if MinLevel(size+1) then
    if x < A[ MaxCousin(size+1) ] then
      TrickleUpMin(x,2,size+1)
    else
      A[ size+1 ]:=A[ MaxCousin(size+1) ]
      TrickleUpMax(x,3,MaxCousin(size+1))
    end if
  else
    if x > A[ MinCousin(size+1) ] then
      TrickleUpMax(x,3,size+1)
    else
      A[ size+1 ]:=A[ MinCousin(size+1) ]
      TrickleUpMin(x,2,MinCousin(size+1))
    end if
  end if

procedure DeleteMin
  x:=A[ size+1 ]
  size:=size-1
  TrickleDownMin(x,2)

procedure MakeDeap(size)
  --- the elements are stored in A[ 2 ]through A[ size+1 ]
  for i:=size+1 downto 2 do
    if MinLevel(i) then
      TrickleDownMin(A[ i ],i)
    else
      TrickleDownMax(A[ i ],i)
    end if
  end for

procedure TrickleDownMin(x,i)
  j:=2*i

```

```

procedure TrickleDownMin(x,i)
  j:=2*i
  while j < size+1 do
    if A[ j ] > A[ j+1 ] then
      j:=j+1
    end if
    A[ j div 2 ]:=A[ j ]
    j:=j*2
  end while
  if j > size+1 then
    j:=j div 2

  if x < MaxCousin(j) then
    TrickleUpMin(x,i,j)
  else
    A[ j ]:=A[ MaxCousin(j) ]
    TrickleUpMax(x,i,MaxCousin(j))
  end if

procedure TrickleUpMin(x,i,n)
  -- this procedure is almost the same as for inserting in a minheap
  for j:= 1 to BinaryMinSearch(x,i,n) do
    A[ n ]:=A[ n div 2 ]
    n:=n div 2
  end for
  A[ n ]:=x

function BinaryMinSearch(x,i,j)
  top:=number of levels between i and j =  $\log_2 j \text{ div } i$ .
  bot:=0
  while top > bot do
    mid:=(top+bot) div 2
    if x  $\geq$  A[ j div 2**mid ] then
      top:=mid
    else
      bot:=mid+1
    end if
  end while
  RETURN top

```

MaxCousin(k) is equal to $k+a$ if $k+a \leq \text{size}+1$, otherwise it is $(k+a) \text{ div } 2$, while MinCousin(k) is $k-a$, where a is $2^{\lceil \lg k \rceil - 1}$. MinLevel(i) is a boolean function that is true if the i :th element is in the MinHeap.

TrickleUpMax, DeleteMax and TrickleDownMax is implemented as the corresponding Min procedures, with some minor changes.

